An investigation into the use of software code metrics in the industrial software development environment

Tim Littlefair

*Edith Cowan University*

Follow this and additional works at: [https://ro.ecu.edu.au/theses](https://ro.ecu.edu.au/theses)

Part of the Software Engineering Commons

**Recommended Citation**


This Thesis is posted at Research Online. [https://ro.ecu.edu.au/theses/1508](https://ro.ecu.edu.au/theses/1508)
An Investigation Into The Use Of Software Code Metrics In The Industrial Software Development Environment

Tim Littlefair
Edith Cowan University

This paper is posted at Research Online.
http://ro.ecu.edu.au/theses/1508
Edith Cowan University

Copyright Warning

You may print or download ONE copy of this document for the purpose of your own research or study.

The University does not authorize you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.

You are reminded of the following:

- Copyright owners are entitled to take legal action against persons who infringe their copyright.

- A reproduction of material that is protected by copyright may be a copyright infringement. Where the reproduction of such material is done without attribution of authorship, with false attribution of authorship or the authorship is treated in a derogatory manner, this may be a breach of the author’s moral rights contained in Part IX of the Copyright Act 1968 (Cth).

- Courts have the power to impose a wide range of civil and criminal sanctions for infringement of copyright, infringement of moral rights and other offences under the Copyright Act 1968 (Cth). Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.
AN INVESTIGATION INTO THE USE OF SOFTWARE CODE METRICS IN THE
INDUSTRIAL SOFTWARE DEVELOPMENT ENVIRONMENT

by

Tim Littlefair B.Sc.

A Thesis Submitted in Partial Fulfilment of the
Requirements for the Award of

Doctor of Philosophy.

At the Faculty of Communications, Health and Science, Edith Cowan
University, Mount Lawley Campus.

Date of submission: 18 June 2001
ABSTRACT

This work describes a project that investigates the use of software metrics based on measurement of source code in industrial software development.

Literature is examined relating to considerations of quality in industry; to software engineering; to emergent technologies in the software industry including the object-oriented paradigm and software patterns; to software product and process metrics; and to cultural factors affecting workplaces in the software industry. It then expounds theoretical and practical work aimed at specifying, designing and implementing a source code analyzer for use in a typical industrial project setting. There is an explanation of the design and outcomes of a questionnaire survey, conducted over the Internet, of developers with an interest in using metrics techniques to analyse source code. Finally, there is a description of a controlled experiment which attempts to detect a benefit from use of source code metrics as information to support a simple software engineering task.
DECLARATION

I certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any institution of higher education; and that to the best of my knowledge and belief it does not contain any material previously written by another person except where due reference is made in the text.

Signature.

Date 26/6/2001
ACKNOWLEDGMENTS

I would like to acknowledge with particular gratitude the assistance of my supervisor, Dr Thomas O’Neill. I am also indebted to a number of other people presently and formerly at Edith Cowan University, including Dr Ken Mullin, Cole Bergersen, and Dr Pender Pedler. Thanks also to Terence Parr, his co-workers and Tom Moog for the excellent public domain PCCTS software, and to many other free software contributors. I am also grateful to Colin Kirsopp and his colleagues at the University of Bournemouth for sharing materials and data relating to their research with me.

I am also grateful to many of my colleagues and to other people on the Internet who helped me by discussing metrics issues, giving feedback on the software released as part of this project, or participating in the questionnaire survey or the code review experiment described in this thesis.

Finally, I would like to thank my partner in life, Kate Eckersley for her loving forbearance during the long period it has taken me to conduct and write up this research project.
TABLE OF CONTENTS

ABSTRACT ...................................................................................................................... ii
DECLARATION .................................................................................................................. iii
ACKNOWLEDGMENTS .................................................................................................... iv

LIST OF TABLES ............................................................................................................. viii
LIST OF FIGURES ........................................................................................................... ix

1 Introduction .................................................................................................................. 10

1.1 Background .............................................................................................................. 10
1.2 Research questions .................................................................................................. 10
1.3 Outline of this document ......................................................................................... 11

2 Review of the literature. .............................................................................................. 14

2.1 Context ...................................................................................................................... 14
2.2 Software engineering ............................................................................................... 14
2.3 The object-oriented paradigm .................................................................................. 16
2.4 Design patterns ........................................................................................................ 19
2.5 Quality ..................................................................................................................... 21
2.6 Software metrics ...................................................................................................... 25
2.7 Cultural context of software development ............................................................. 29

3 Theoretical considerations .......................................................................................... 34

3.1 Background .............................................................................................................. 34
3.2 Selection of measures for implementation .............................................................. 34
3.3 Goal/Question/Metric Analysis ................................................................................ 36

3.3.1 Goal selection ..................................................................................................... 38
3.3.2 Question development ......................................................................................... 39
3.3.3 Metric selection ................................................................................................... 58
3.4 Theoretical bases of validation ................................................................................ 67

3.4.1 Observations from measurement theory ............................................................ 69
3.4.2 Weyuker’s properties of software complexity measures ..................................... 72
3.4.3 Applying Weyuker’s complexity axioms over an extended model of metric use ........................................................................................................... 75
3.4.4 A data model for measurement of object-oriented software ................................. 79
3.4.5 Evaluation of selected metrics using Weyuker’s framework and the extended model ........................................................................................................ 83

4 Implementation of a metric tool .................................................................................. 92

4.1 Background .............................................................................................................. 92
4.2 Chronology of the analyzer implementation ............................................................ 92

4.2.1 Initial specification ............................................................................................. 92
4.2.2 Prototype implementation .................................................................................. 93
4.2.3 Intermediate versions ......................................................................................... 94
4.2.4 The reference version ......................................................................................... 96
4.3 Architecture of the analyzer .................................................................................... 97

4.3.1 The command line processor ............................................................................ 98
4.3.2 The language recognition front ends ................................................................ 98
4.3.3 The internal database ....................................................................................... 102
4.3.4 The output module ............................................................................................ 105

4.4 Data collection algorithms for metrics ..................................................................... 106
4.4.1 Calculation methods for procedural measures ................................................. 108
4.4.2 Calculation methods for structural measures .................................................... 111
4.4.3 Calculation methods for Chidamber and Kemerer's measures .................................................... 113
4.5 Potential enhancements identified .................................................. 113

5 Practitioner opinion survey .................................................. 115
  5.1 Background .................................................. 115
  5.2 Goals of the survey ........................................ 115
  5.3 Factors under investigation ........................................ 116
    5.3.1 Input factors ........................................ 116
    5.3.2 Outcome factors ...................................... 117
  5.4 Survey Design ........................................ 117
  5.5 Survey administration and processing .................................. 119

6 Survey outcomes .................................................. 121
  6.1 Background .................................................. 121
  6.2 General questions ........................................ 123
    6.2.1 Main source of income .................................. 123
    6.2.2 Employment status ..................................... 124
    6.2.3 Years of experience .................................... 124
    6.2.4 Summary for general questions ......................... 125
  6.3 Binary questions ........................................ 125
    6.3.1 Responsibilities ....................................... 126
    6.3.2 Languages, methodologies and techniques ............... 126
    6.3.3 Application areas and platforms ....................... 128
    6.3.4 Summary for binary questions .......................... 130
  6.4 Agreement questions ...................................... 131
    6.4.1 Preferred ways of working ............................. 131
    6.4.2 Professional and social environment .................... 132
    6.4.3 General metrics issues ................................ 133
    6.4.4 Issues relating to procedural metrics .................. 135
    6.4.5 Issues relating to structural metrics .................. 136
    6.4.6 Issues related to object-oriented metrics ............... 138
    6.4.7 Issues relating to the CCCC tool ...................... 139
    6.4.8 Summary for agreement questions ....................... 141
  6.5 Responses to the free format question ................................ 142
  6.6 A comparison survey ........................................ 142

7 Code Review Experiment .................................................. 146
  7.1 Background .................................................. 146
  7.2 The conception of the experiment .................................. 147
  7.3 Design of the review exercise .................................. 150
  7.4 Plan for processing of data ................................... 154
  7.5 Recruitment of subjects ...................................... 155
  7.6 Statistical terminology and conventions ....................... 162
  7.7 Raw data .................................................. 165
  7.8 Derivation of correct responses ................................ 167
  7.9 Categorization of responses ................................... 168
  7.10 Receiver Operating Characteristic Analysis .................... 169
  7.11 Chi-squared analysis ...................................... 173
  7.12 Additional investigations ................................... 176
  7.13 Summary of the experiment ................................... 177

8 Conclusion .................................................. 178
  8.1 Findings in relation to the initial research questions .............. 178
    8.1.1 Feasibility of implementation ............................. 178
    8.1.2 Theoretical validation ................................... 179
    8.1.3 Practical validation ..................................... 179
    8.1.4 Interface ergonomics ..................................... 181
    8.1.5 Breadth of application ................................... 182
  8.2 Other findings ........................................ 182
    8.2.1 Novel metrics ........................................ 183
8.2.2 Use of pragmatic measurement techniques ............. 184
8.3 Suggestions for further work ...................................... 184
  8.3.1 Further development of the analyzer .................. 184
  8.3.2 Consideration of structural metrics ................. 185
  8.3.3 Further statistical analysis of survey data ........ 185
  8.3.4 Use of the analyzer to support research in other areas .
  ................................................................. 185
  8.3.5 Further empirical work .................................... 186
Appendix A: Specification for the prototype analyzer .......... 188
Appendix B: Sample Run Input and Output ......................... 190
Appendix C: User Guide for CCCC ................................. 223
Appendix D: Analysis of download logs for the project FTP area 231
Appendix E: Questionnaire used for practitioner survey ......... 233
Appendix F: Raw survey data ......................................... 238
Appendix G: Responses to free-format question on survey .......... 242
Appendix H: Materials from the Code Review Experiment ........ 249
Appendix I: Statistical Worksheets ............................. 258
List of References .............................................. 265
# LIST OF TABLES

Table 1: Greek and Roman culture .................................................. 32  
Table 2: Effect of abstraction on change propagation ..................... 53  
Table 3: Relationships between Metrics and Quality Model Attributes .................................................. 67  
Table 4: Which metrics satisfied which of Weyuker's axioms ........... 91  
Table 5: Source of income ............................................................ 123  
Table 6: Employment status .......................................................... 124  
Table 7: Length of experience ...................................................... 124  
Table 8: Responsibilities ............................................................. 126  
Table 9: Languages, techniques and methodologies ......................... 127  
Table 10: Application areas and platforms .................................. 129  
Table 11: Preferred ways of working .......................................... 132  
Table 12: Professional and social environment ................................ 133  
Table 13: General metrics issues ............................................... 134  
Table 14: Issues relating to procedural metrics ............................ 136  
Table 15: Issues relating to structural metrics ............................. 137  
Table 16: Issues relating to object-oriented metrics ..................... 138  
Table 17: Issues relating to features of CCCC .............................. 140  
Table 18: Initial volunteer statistics ......................................... 159  
Table 19: Final volunteer statistics ............................................ 159  
Table 20: Raw Data from Code Review Experiment .......................... 165  
Table 21: Cumulated Responses of Each Group ............................. 167  
Table 22: Derived Correct Responses ........................................... 167  
Table 23: Categorization of Individual Responses .......................... 168  
Table 24: Categorization of Group Responses ................................ 168  
Table 25: Contingency table for groups 1 and 2 ............................ 174  
Table 26: Chi squared contributions for each contingency .............. 175  
Table 27: Raw data from questionnaire survey .............................. 239  
Table 28: Derivation of Correct Responses .................................... 259  
Table 29: ROC Analysis for Group 0 ............................................ 261  
Table 30: ROC Analysis for Group 1 ............................................ 262  
Table 31: ROC Analysis for Group 2 ............................................ 263  
Table 32: Chi-squared analysis .................................................... 264
LIST OF FIGURES

Figure 1: Boehm's Model of Quality ......................... 38
Figure 2: ROC curves for groups 0, 1 and 2 ................. 172
1 Introduction

1.1 Background

Software metrics have been proposed as potential tools in the endeavour to improve the quality of computer software, and to help in the control of software projects. The research described in this document focuses on the use of metrics in the industrial software development situation. The project concentrates particularly on that class of metrics that can be automatically calculated from software source code, and on the context of use and usefulness of features of an automated tool for gathering of source code metrics.

While the literature research in Chapter 2 shows that there has been a fair amount of discussion on different source-based metrics, the writings contain relatively little evidence on the value or otherwise of these techniques in an industrial software engineering setting. The research questions for the current project are formulated with the intent of focussing investigations in this under-researched area.

1.2 Research questions

The research questions as defined in the project proposal were as follows:

- Can we implement tools to measure the selected metrics in an economical way, and gather data using those tools without negative impacts on the development process?
- Do the metrics meet the theoretical requirements for software metrics proposed in the literature?
• Are the metric results of practical value in an industrial software development setting?
• How can we design the interface through which metric tools report back results to improve the quality outcomes of the development process?
• How general are the lessons learned in this study? Can they be applied in situations involving other metrics, or to organizations which have different operational contexts?

1.3 Outline of this document

Following this introduction, chapter 2 of this thesis consists of a review of literature in a number of areas relevant to the practical work undertaken. The review considers:

• literature on the topic of software engineering;
• general management literature on quality management;
• literature specifically relating quality management to software development; and
• literature concerning both software metrics in general and particular proposed measures.

Literature on a variety of proposed metrics is explored, but emphasis is placed on metrics which can be conveniently measured by automated means (usually by running some kind of analyzer program over a body of source code in a programming language). These source code related metrics include measures of size, such as the number of lines of code, and measures which purportedly relate to source code quality, including some which relate to aspects of structuredness, readability,
understandability, and the use of the object-oriented programming paradigm.

Following the literature survey, chapter 3 amplifies and extends some of the theoretical issues raised in the literature. This chapter seeks to define the theoretical basis of the work undertaken in the practical development section of the research project. In particular, there is a description of the basis on which a number of metrics from the literature were selected for implementation in an automatic measurement tool. The chapter also includes discussion of the relationship between the definitions of the metrics implemented and the basis of calculation applied for each such metric. Descriptions are presented of a small number of metrics which, while based on work described in the literature, demonstrate significant novel modifications proposed to generate metrics that are tailored to reflect quality factors of increasing importance in object-oriented projects.

Chapter 4 describes the technology used to implement the metric analyzer tool, including an informal description of the initial structural design and historical material on the facilities which were added or changed in the course of field testing the tool.

Chapter 5 describes the design and administration of a survey of opinions on software metrics issues completed by a number of software engineering practitioners. The survey questionnaire is designed to obtain the following data:
• factual information about the respondent (years of experience, areas of responsibility at work, specific languages/tools/methodologies in use);
• attitudinal information about the respondent's preferred ways of working;
• information about the respondent's working environment;
• the respondent's opinions about the interpretation and value of different groups of metrics; and
• the respondent's opinions on specific features of the analyzer tool and the novel measures presented.

Chapter 6 presents the outcomes of the survey described in the previous chapter.

Chapter 7 presents the design and outcomes of an experiment, which attempts to detect an effect from the use of metrics information in a simulated software engineering task.

Chapter 8 concludes the thesis by revisiting the main findings of the theoretical, development, survey and experiment stages of the research, and raising issues which are likely to be fruitful areas for future study.
2.1 **Context**

The introduction of quality management ideas into the software industry reflects the interaction of a number of streams of managerial theory, some specific to the software industry itself, and others applying to a broader range of activities. These different streams can probably be categorized as coming from two separate initial sources. The first is the relatively recent discipline of Software Engineering, which consists of ideas about the software industry developed since the early 1970s. The other is general quality management theory, which has its roots in work done in Japanese industry since the Second World War. Although the general stream of quality thinking is older, its popularity outside Japan is more recent, and it is only in the last ten years or so that it has been commonly discussed in the literature of software management.

2.2 **Software engineering**

The development of the discipline of Software Engineering began with the realization, by leading figures in the computing industry around the late 1960s that existing methods of management were not providing successful control of software projects.

The uncontrolled or ill-controlled state of industrial software activities from which the early software engineering pioneers were seeking to escape was frequently referred to as "the software crisis". F.P. Brooks' book *The mythical man-month* (1975) is the classic description of a large software project out of control.
Writers in the software engineering area have proposed a number of changes to the way software is developed and used to make these processes more manageable. One of the most important early writers in the area is Barry Boehm, whose major contribution is a body of work on the prediction of resource requirements for projects of varying sizes, which is known as the Constructive Cost Model, or COCOMO (Boehm, 1981).

When practitioners including Boehm began to gather and interpret data on the software development processes in their organizations, two important observations were recorded:

- that the repair of defects consumes 50% of the total labour resources expended in the life cycle of a typical software project (Boehm & Papaccio, 1990); and

- that small defects (particularly if they are introduced early in the product life cycle) could cause large costs to the organization if they were not eliminated as soon as possible after their introduction (Ward, 1991).

These empirical observations have led to a realisation that one way of reducing the lifetime cost of a project might be to focus more resources on achieving high-quality results in the earlier phases of the life cycle. Changes which have been suggested include:

- use of inspection or review techniques to eliminate defects at an early stage, as proposed by Fagan (1976) and Yourdon (1979a);

- use of top-down structured analysis, design and implementation methods, often proposed in the context of a prescribed documentation methodology such as Jackson Structured Development (King, 1988),
Structured Analysis/Structured Design (Yourdon, 1979b), or Structured Systems Analysis and Design Method (Longworth, 1989); and

- use of object-oriented techniques in analysis design and implementation (these will be covered more fully in the next section).

### 2.3 The object-oriented paradigm

In the course of the 1980s and early 1990s, the object-oriented paradigm emerged and attracted increasing support throughout the software engineering profession. This technology originated from developments in new computer languages, particularly the language Smalltalk (Goldberg & Robson, 1983).

The object-oriented paradigm proposes the use of hierarchically-related abstract data types to implement a system as a collection of highly independent subsystems. These interact with one another through small, carefully designed, interfaces. Each subsystem is unable to access implementation details of other subsystems. The management of visibility in this way is called 'information hiding' and is said to yield designs which have lower degrees of coupling between different subsystems. Lower coupling is said to translate into greater ability to implement the separated systems independently of one another. It also reduces maintenance effort as the implications of modifications to any part of the implementation are less likely to cascade through other parts of the system requiring widely distributed changes.

The object-oriented paradigm also provides for the definition of new classes of objects that inherit much of their behaviour from previously defined classes. Inheritance is said to favour re-use of standardized
software components. Ledbetter and Cox (1990) refer to such re-usable components (using their favoured language Objective-C) as Software-ICs. This name reflects the hope that just as integrated circuits (ICs) have revolutionized electronic hardware design by providing a wide range of off-the-shelf subsystems that can be assembled for new applications, similar benefits can be obtained in software by applying re-usable object-oriented software components.

Frederick Brooks quotes Mark Sherman's observation that information hiding and hierarchical typing are, in fact, mutually orthogonal language features (Brooks, 1990, p. 19). Wegner (1987) provides an extensive review of the options raised by the object-oriented paradigm from the point of view of the language designer, listing objects, classes, inheritance, data abstraction, strong typing, concurrency and persistence as features which may need to be supported (either in the language or in libraries). The interest of these observations in the current study is that they may provide hints as to the attributes of the source code of object-oriented programs that we may wish to measure.

In the course of the 1990s, object-oriented concepts became part of mainstream language technology, and there is now a range of languages and products which offer features to support object-orientation as an increment to mature mass-use languages. Objective C, as mentioned above is an example of this, but probably the most important example is the language C++ (Stroustrup, 1988), which added facilities for user defined types and polymorphism to the successful language C. Since its initial appearance in the early 1980s, C++ has undergone considerable evolution, including the introduction of exception handling and parameterized types,
which are described in The annotated C++ reference manual (Ellis & Stroustrup, 1990), and which are now widely supported in compilers and other language-related tools.

In 1998 an ISO standard for the language was published as ISO/IEC Standard 14882-1998. The standard adds a number of syntactic features into the language, but is upwardly compatible with previous versions of the language (as defined by the specifications released by Stroustrup’s group at AT&T who were responsible for the original development). The history of the language, together with the reasoning behind many of the steps in its evolution is described by its principal designer in (Stroustrup, 1994).

With the increasing use in mainstream software development of languages that are able to support the object-oriented paradigm, there has been an emergence of methodologies for analysis, design and implementation of projects based on these new ideas. Two of the most widely practiced such methodologies are the Booch methodology (Booch, 1994) and the Object Modelling Technique (or OMT) methodology proposed by James Rumbaugh and others (Rumbaugh, Blaha, Premerlani, Eddy & Lorenson, 1991). A third influential methodology is called Objectory, pioneered by Ivar Jacobson, which includes the concept of Use Case modelling (Jacobson, Christerson, Jonsson & Overgaard, 1992). Booch, Rumbaugh and Jacobson are all now associated with the software tool vendor Rational Corporation, and are working on harmonizing their approaches. A draft description of a proposed Unified Notation for Object-Oriented Development has been published (Booch & Rumbaugh, 1995).
2.4 Design patterns

An interesting emerging area of research leading on from object-oriented development is the pattern language movement. Languages such as Smalltalk and C++ provide encapsulation by class and inheritance as the primary language-sponsored means of modularization. There are, however, large numbers of useful re-usable abstractions which cannot be effectively expressed in the form of a single class. Grady Booch observes:

It is important to note that the class - as defined by most programming languages - is a necessary but insufficient vehicle for decomposition. Sometimes abstractions are so complex that they cannot be conveniently expressed in a single class declaration.

For example, at a sufficiently high level of abstraction, a GUI framework, a database, and an entire inventory system are all conceptually individual objects, none of which can be expressed as a single class. Instead, it is far better to capture these abstractions as a cluster of classes whose instances collaborate to provide the desired structure and behaviour. Stroustrup calls such a structure a component.

(Booch, 1994, p. 104).

Booch's methodology introduces the term 'class category' to denote a component of a software design containing either a cluster of cooperating classes, or at the higher levels a cluster of nested class categories.

James Coplien, who was one of the earliest users of C++ within AT&T (Stroustrup, 1994, p. 66), was responsible for a book called 'Advanced C++ Styles and Idioms' which discusses a wide variety of these clustered
abstractions (Coplien, 1992). Many of the techniques he presents have names, including:

- orthodox canonical class form (ibid., p. 38);
- envelope and letter classes (ibid., p. 70);
- handle and body classes (ibid., p. 62); and
- functors (ibid., p. 165).

Coplien has since gone on to become one of the leading lights of the pattern language community of software theory. The pattern community builds on ideas presented in the literature of architecture and planning in the 1960s by Christopher Alexander (Coplien & Schmidt, 1996, p. xi). While Coplien’s earlier book predates the pattern movement in computer science and does not use the term, the advanced techniques he describes fit well into the formalism of patterns. Each technique has a name, identifies a set of recurring problems, and defines a structure that provides a generic re-usable solution to those problems.

A landmark in the development of software pattern thought was the publication of the book Design patterns: elements of re-usable object-oriented software (Gamma, Helm, Johnson & Vlissides, 1995). This book presents a catalogue of widely occurring patterns.

Robert C. Martin is a software practitioner and author who has written on topics relating to design at the class category level. His work includes a description of the process of examining the design of a large suite of programs looking for evidence of recurring patterns:
After we had become familiar with the concept of patterns, we decided to reexamine the design and implementation of the first program and framework to see if we could describe any patterns. Not surprisingly, we found quite a few. In fact, in one form or another, every pattern in Gamma et al.'s 'Design Patterns' was used in our application, yet in very ad hoc and vague ways. Had we known about these patterns to begin with, I believe we would have spent less time searching for solutions and would have created our designs with more discipline and structure.


Given the current interest in patterns, there is some interest in using metrics to automate the process of pattern identification in existing software (Kim & Boldyreff, 1997).

2.5 Quality

In parallel with the development of software engineering practices, the past ten years have seen the increasing application by large corporations of various flavours of quality management. Gillies (1992, p. 129) identifies W. Edwards Deming, Joseph Juran and Philip Crosby as key 'gurus' whose teachings on quality are influential across a wide range of companies.

The models of quality presented by the different thinkers (and other contributors in the same area) are by no means identical. One common thread present in most if not all of their prescriptions is the pervasive use of measurements and numerical process monitoring to achieve consistency in the process and to help identify and eliminate the root causes of defects (ibid., pp. 132-33). Other common prescriptions include:
• the need for quality management to be instituted on a company-wide basis, with leadership from top management (Juran, 1988, pp. 244-246); and
• the need for a cyclical model of the improvement process which incorporates feedback from information being gathered in the present into the planning process for the future (Gillies, 1992, p. 135).

Deming emphasizes the proposition that data gathered for the purpose of process improvement should not be available for purposes related to assessment of the performance of staff, as this is likely to lead to manipulation of the data gathering process to yield politically acceptable results (Deming, 1986, p. 264). This concern mirrors ideas expressed by several of the software engineering writers who propose review or walkthrough methods: e.g. Fagan (1976, p. 197).

The spread of quality management ideas is presently beginning to impact upon companies who have not voluntarily adopted them. This is particularly due to the increasing requirement of large purchasers of equipment and services (particularly in the government sector) to demand that suppliers have a quality management system in place and accredited by a third party, often to standards of the ISO 9000 series (Gillies, 1992, p. 170).

Recent literature in the software engineering field is beginning to work toward a synthesis of the ideas of quality management with those of software engineering. The work by Gillies (1992), is a broad survey of the interaction of the two fields of endeavour. This book summarizes work by various writers including Gilb, Boehm and McCall who attempt to answer the question "What do we mean by quality in software?" These writers
typically propose hierarchical models of quality, with high-level concepts like maintainability, utility and efficiency broken down into lower level attributes like structuredness, accuracy, and self-containedness (ibid., p. 21-27).

Gilb's work (1988) proposes that a multidimensional model of quality should be created as part of the specification for each project. This model should include explicit objective measures of each attribute, and that it should be refined over time throughout the project by consultation with the project customer, in a process referred to as evolutionary delivery. Other writers have also endorsed similar models of the software life-cycle, which are variously referred to as prototyping, or spiral models (Gillies, 1992, p. 108; Boehm & Papaccio, 1990, p. 39).

An early taxonomy of maturity was proposed by Meilir Page-Jones. This scheme, known as the 'ages of software', classifies the maturity of a particular development environment as indicated in the stages listed below:

Some groups operate in the age of anarchy, developing software without the benefit of any systematic approaches or even codified wisdom. Everything rests on the skill of the individual. The age of folklore is classified by a culture of collective wisdom, accumulated knowledge that is often embodied in stories about successes or failures or rules of thumb extracted from past experiences. The age of methods is based in systematic, although not necessarily formal, approaches to software development that go beyond folklore. The age of metrics is based on measures for evaluating quality and productivity and organized feedback for improving the development process based on measurement. Finally, we
reach the age of engineering, in which software development becomes a true engineering discipline, a process under continuous improvement using methods which are not based in folklore or armchair speculation but on theory validated through study and research, in which design decisions and trade-offs are systematic and derived from models and metrics that embody the results of a growing body of knowledge. (Constantine, 1995, p. 51).

Another influential writer is Watts S. Humphrey, who is responsible for work on the software process maturity framework known as the Capability Maturity Model, sometimes abbreviated CMM (Humphrey, 1989, p. 6; Humphrey, 1990). This model characterizes organizational software development processes into five levels:

- initial (referred to by some writers as 'chaos');
- repeatable;
- defined;
- managed; and
- optimizing.

Humphrey claims that improvement of the software process must go through these phases, and that attempts to skip one or more phases are likely to be met with failure (Humphrey, 1989, pp. 5-6). The Humphrey's CMM framework has become an orthodoxy of the software industry, although there is some dissent over the proposition that maturity can only develop by progressing through the levels one at a time. For example, Robert L. Glass, discussing research reported by Stan Rifkin in the early 1990s writes:
They seem to have achieved a process level of 4 or 5 without ever having passed through level 3. That is, they measure and use feedback, but have not invoked a defined process for developing software. (Glass, 1994, p. 9).

### 2.6 Software metrics

From the earliest days of the software engineering discipline there has been wide agreement of the need to measure software processes and products as a pre-condition for establishing control over development activities.

One of the first and simplest software product metrics is the number of lines of source code (Conte, Dunsmore & Shen, 1986, p. 34). While there are a number of conflicting definitions of a line of code (or of the very similar measure 'delivered source instructions'), there are a few features of this attribute which make its early appearance in the field unsurprising. These features are, firstly, that automated tools to measure lines of code are easily implemented; and, secondly, that the results of the measurement (given a specific definition) are intuitively meaningful to personnel involved with software. The second of these features has the desirable consequence that experienced staff on a project are likely to be able to produce reasonable estimates of the amount of code required before the code is written (Humphrey, 1989, p. 20).

Other examples of seminal early product metrics include Halstead's Software Science, described in Conte et al. (1986, pp. 36-42), and McCabe's Cyclomatic Complexity (McCabe, 1976). Boehm's COCOMO model of the software development process (Boehm, 1981) is a good example of a
collection of metrics relating to processes and resources designed to aid in the prediction of schedule requirements of large projects.

Measurement theory categorizes measures in general (including software metrics) as being divisible into direct and indirect measures:

Direct measurement of an attribute is a measure which does not depend on the measurement of any other attribute. Indirect measurement of an attribute is measurement which involves the measurement of one or more other attributes. (Fenton, 1991, p. 18).

Lines of code and Halstead's token counts are good examples of direct measures. McCabe's cyclomatic number can be viewed as a direct measure of the number of independent paths through the procedural logic of a subprogram, but it is more commonly viewed as an indirect measure of maintainability, and particularly of testability (McCabe, 1976, p. 318).

Another commonly described categorization of software metrics is into product and process metrics (Conte, et al, 1986, p.19). Product metrics are measurements which are applied to one or other of the products created in the process, including source code, the final executable program, analysis or design documentation. Process metrics are used to characterize the development process itself, and may measure features like human effort expended over the whole process or any particular phase, number of defects found, or cost to fix defects.

Various writers have tackled the issue of desirable qualities of a software metric. Conte et al (ibid., pp. 21-22) list simplicity, validity, robustness, prescriptiveness, and analyzability, but note that
these attributes are not all easily defined objectively. Fenton (1991) emphasizes the use of the representational theory of measurement and empirical validation of proposed metrics. The framework suggested by Weyuker (1988) for evaluation of metrics relating to software complexity, suggests nine desirable properties, including:

- that different programs can conceivably yield the same value for the complexity measure;
- that different implementations of an identically specified function are capable of yielding different complexity values; and
- that a program composed from two interacting subprograms should yield a higher complexity measure than the sum of the measures of the subprograms individually.

The literature includes a large number of product metrics based on analysis of source code, presumably because the implementation of such metrics is easily automatable. As well as McCabe’s and Halstead’s work mentioned above, there are contributions including Ejiogu’s tree methodology (Ejiogu, 1991), Ada package metrics (Gannon, Katz & Basili, 1986), module coupling metrics (Offutt, Harrold & Kolte, 1993), and coupling measures for C++ (Rajaraman & Lyu, 1992).

Some metrics are also proposed as applicable to designs rather than implementation source code, but are similar in style and lend themselves to implementation by parsing source code. This group includes the metrics suite suggested by Chidamber and Kemerer (1991), as well as those proposed by Chen and Lu (1993), both of which cover attributes relating to the object-oriented paradigm.
One criticism of metrics, which are calculated from source code, is that they only become available at a late stage in a project's life-cycle, and are, therefore, of little value in the crucial early phases of a project when initial planning is taking place. To fill this need, the function points metric has been proposed (Albrecht & Gaffney, 1983).

Although function points are derived from the specification for a project, and hence are available earlier, they require some subjective judgement to count. For this reason the creation of automated tools to gather the metric is problematical, firstly because of the subjectivity of the counting process, and secondly because of the lack of a widely used formal language for the expression of specifications. An experiment which found a reasonable degree of consistency in function point counts between trained personnel is presented in Kemerer (1993), while a criticism of the original function points work together with proposed modifications appears in Symons (1990).

It has been noted that the accepted models of software quality tend to be multidimensional and hierarchical. It follows from this that no single metric is ever likely to be presentable as a single expression of software quality. To make metrics useful in a practical industry setting, it is necessary to identify the significant quality attributes of the current project, which can then be used to derive or select useful metrics. This process is formalized in the goal/question/metric paradigm, proposed by Basili and Rombach (1988). Gilb's evolutionary development (1988) employs similar ideas, but is much wider, organizing the entire development process around the identification, specification and pursuit of numerically specified quality attributes.
2.7 Cultural context of software development

One of the goals of the current study was to investigate the interaction between personal, cultural and organizational factors and the use of software metrics based on source code analysis. The early chapters of Robert L. Glass's Software Creativity (Glass, 1995), describe various perspectives on the tension between the requirement for personal freedom implied by the intellectual challenge of software design and construction, and the requirement for corporate discipline implied by the economic organization of the software industry. He observes:

Companies seem to have a corporate culture that emphasizes either of the following:

Management, with the goal being to control in order to improve.

Technologists, with the goal being to experiment in order to improve.

(ibid., p. 8)

and later:

[In a study published by the Software Engineering Institute, SEI document CMU/SEI-91-TR-16] ... Rifkin and Cox came to the conclusion that some companies are management control driven and others technologist/experimentation driven ... ... Rifkin and Cox clearly found that the companies which emphasized technology and experimentation were the leaders in the successful use of metrics. 

... They [the technologist driven companies] tend to have achieved SEI process level of 4 or 5 (very good) without ever having passed through level 3. That is, they measure and use feedback to improve their software process without ever having invoked a defined process! ... ... Measurement is 'part of how we do business'.

29
Glass links the conclusions of the Rifkin-Cox study to a simultaneously published study of software practices by DeGrace and Stahl, which identified two canonical software cultures, labelling them 'Greek' and 'Roman' by analogy with the ethos, strengths and weaknesses of those two ancient civilizations:

In ancient Greece, an individual would act as an agent in his own behalf, or combine with other people to act together as a team. In a Greek work environment, you bring your tools to work with you, you do your stuff, and then you pack up your tools and take them home. You are an individual ... an independent contractor. You are not owned body and mind. You merely providing a service for compensation.

In Rome, one’s first duty was to the group, clan, class or faction upon which one depended for status. Known as ‘gravitas’, this meant sacrificing oneself for the good of the organization, and giving up one’s individuality and identifying closely with the group. In a Roman environment, you go to work, the company hands you your tools, and then holds you and your mind hostage until you sever your relationship with the organization. You are not an individual: you are owned by the organization body and mind, twenty four hours a day. There are substantial rewards for this, however. The organization provides you with security, money and power.

( Ibid., pp. 8-9).

While the Rifkin-Cox study sees whole companies as belonging on one side or the other of the divide, the DeGrace/Stahl work acknowledges that it is likely that both cultures often coexist within the same organization:

... In our opinion, the appropriate roles are that the Romans provide a working context in which the Greeks get to do their
stuff and express their results in formal terms for the Romans.

(ibid., p.6)

The De Grace/Stahl work is cited by Glass as presenting a catalog of differences that can be used to distinguish between Greek and Roman culture. This catalog is reproduced as Table 1.
The relevance of literature in this area to the current research project is related to the recognition in the research questions that the measurement techniques being investigated do not operate in a vacuum. Source code based software metrics are proposed as useful measures for
deployment in workplaces where software engineers are employed to perform
development, and their effectiveness may be affected by issues relating
to the varying environments in different such workplaces. The Greek/Roman
cultural dichotomy discussed by Glass has been selected as the primary
framework for describing workplace differences in the practitioner survey
that forms part of the current project.
3 Theoretical considerations

3.1 Background

This section will discuss the theoretical considerations which apply to various parts of the project including:

- selection of a set of metrics which can usefully be applied in a given industrial setting;
- development of new metrics characterizing desirable features of modern programming style (principally aspects of the object-oriented paradigm); and
- considerations relating to measurement theory approaches to the validation of software code metrics.

3.2 Selection of measures for implementation

The intention of this project was to investigate the practical use of software code metrics in an industrial software development environment. In order to obtain realistic information about the context in which the metrics are used, it was planned that an industry partner would be recruited, through whom the researcher would be able to work with a project team.

Some difficulty was experienced in finding a suitable industry partner. After a period of attempting informal contacts with perceived likely candidate companies (of whom there were relatively few in the geographically isolated market of Perth, Western Australia), the researcher and supervisor agreed to approach the researcher's full time employer. The employer agreed that the researcher should be allowed to
work with a project team building a large real time system in C++. The size of the team varied from 8 to 16 over the 12 month period of this phase of the research, and for all of this time the researcher was a member of the team.

The presence of the researcher as a member of the subject team was far from ideal; it presented a likely mechanism for the magnification of the danger that the evaluation survey would be positively biased in order to please the researcher. It was for this reason that some aspects of the original project plan were modified at this stage. It was decided that the project team would be treated as an example test site only, and that the main deployment and evaluation exercises would be performed via the Internet, with users who had no face-to-face contact with the researcher.

The researcher had access to the project team over a period of more than twelve months, and was able to undertake the following activities:

- observation of the team's progress and of issues which appeared to impact on progress and product quality;
- discussion, particularly with the principal members of the team, of the value or otherwise of various proposed metrics; and
- field experiments with the metric analyzer on source code developed by the team.

Although the members of the project team were broadly supportive of and interested in the research topic, and gave generously of their time to discuss metrics issues and review test versions of the analyzer software, these activities were neither formal nor structured. It would be fair to say that the feedback obtained could be characterised as being closer to
'polite encouragement' than 'assertive guidance'. While every attempt was made to enable the team to participate in the selection of metrics for investigation, the decisions taken in this area were all made by the researcher, albeit with a conscious effort to address the requirements of the project, as opposed to any personal agenda.

3.3 Goal/Question/Metric Analysis

The literature review chapter briefly mentioned the goal/question/metric paradigm, and the related issue of quality models for software projects. The premise of goal/question/metric (or GQM) is that the most appropriate framework for application of metrics is in the context of an analysis that contains the following steps:

- Identification of 'goals' which are abstract attributes that the person undertaking the analysis regards as desirable in a software process or product. Goals are usually abstract nouns, e.g. testability, structuredness, portability, maintainability.
- Posing of 'questions' which express, as an interrogative sentence in a human language, the dichotomy between the presence and the absence of the goal attribute. An example question would be 'Has the system been decomposed into understandable procedural modules?'.
- Defining of 'metrics' which are procedures, formulae or algorithms that may be presented as evidence in reply to the question. Example metrics might include mean number of non-comment lines of code per function, mean number of parameters per function, etc.

The outcome of the GQM analysis is a three-layer network of goals, questions and metrics. The network contains forward feeding links representing the outcomes of the GQM analysis itself in terms of
proceeding from vaguely defined goal attributes through revealing questions to identifying useful metrics. It also contains backward feeding links showing the relationship of the measurements made back through the questions that they answer to the high-level goal attributes. Among the forward links, each goal may give rise to more than one question, and each question may in turn suggest more than one metric. For the reverse links, each metric may contribute to the answer of more than one question, but each question is likely to be closely bound to a single goal.

The questions generated have a pivotal role in the GQM process. The answers to the questions will tend to have a relative, rather than an absolute, form. For example, in response to the example question 'Has the system been decomposed into understandable procedural modules?', rather than a simple 'yes' or 'no' response one might expect a response that gave some kind of ranking to the decomposition of different systems or different parts of the same system. Based on this ranking the person responsible for managing the development effort might do any of the following things:

- estimate resources required for future development or for maintenance based upon past experience of the relationship of measurement outcomes to ongoing resource requirements;
- identify some systems or components with unfavourable ranking outcomes for review or rework;
- lay down standards for acceptance of systems or components at particular life-cycle milestones based on ranking thresholds; and/or
- do nothing.
3.3.1 Goal selection

For the current project, the goals were selected based on a hierarchical model of software development quality proposed by Boehm, as presented by Gillies (1992, p. 25). The base model is illustrated in Figure 1:

Of the low-level quality attributes mentioned in this diagram, the following were selected as being at least partially reflected by some aspect of product source code: self-containedness; self-descriptiveness; structuredness; conciseness; legibility; and augmentability.
The quality model illustrated in Figure 1 was originally developed in the period before the widespread acceptance of the object-oriented paradigm as a significant software technology. In the context of the present investigation, it was decided to add low-level quality attribute goals to reflect the fact that the reference team was using object-oriented techniques and tools in both the design and implementation of its system. The added attributes were: abstractness; reusability; and buildability.

3.3.2 Question development

For each of the low-level quality attributes listed in the preceding section, one or more questions were posed which are intended to express the way the attribute in question is seen as contributing to the quality goals of the project. The questions were mostly posed in relation to the attributes of the individual modules which make up the system under development, although a few are applicable to the system as a whole. Each question was assigned an identifier to indicate the quality attribute from which it was derived.

3.3.2.1 Questions relating to self-containedness

- SELFCONT1
  How much use must be made of other modules to exercise the public interface(s) of a module?

- SELFCONT2
  How much use must be made of other modules to provide the implementation of a module?

- SELFCONT3
  How many other modules contribute behaviour to the interface of the current module (i.e., via inheritance)?
From the initial quality model diagram we see that Boehm expects self-containedness to contribute to the higher level quality attributes portability and reliability. The reasons for this are straightforward: to the extent to which a module is not self-contained, it has an inherent dependency for its availability and correct operation on the availability and correct operation of one or more other modules. The dependent module may be rendered incorrect or unavailable by a change in the availability or behaviour of the dependee.

The desirability of self-containedness does not imply that modules should be completely self-contained, but rather that the numbers and degrees of dependencies between modules should be monitored as a source of project risk. Some kinds of dependencies create risks of worse consequence than others. If a dependency is localized in the implementation of a module, we can be reasonably sure that rework required due to a change in the dependee can (theoretically) also be localized to the implementation. If a dependency is expressed in the public interface of a module, then change to the dependee may well cascade through the current module into any or all of the modules that depend upon it.

The risks associated with dependencies also vary with the closeness of the relationship between the two modules in terms of the overall system's structure, and with the dependee's expected stability. If the dependent and dependee class are being developed by the same developer and implement closely related areas of the overall problem domain, it is likely that changes to the dependee module will be made with special attention to their impact on the dependent module. Dependencies that reach into different partitions of the overall system are more likely to
import unpleasant surprises, particularly if the dependee is owned by a
different programmer. On the other hand, modules that are widely used
across the system may have lower risks for the programmer responsible for
the dependent module, although wide usage may reduce the degree of
flexibility available to the programmer responsible for the dependee,
especially if it is still under active development.

3.3.2.2 Questions relating to self-descriptiveness

• SELFDESC1

Does the source code for the component contain an appropriate amount
of meaningful comment?

Widespread use of comments has been recommended practice for programmers
since the development of the earliest programming languages. While it is
difficult to envisage automation of measurement that takes into
consideration the meaningfulness of comments, it is straightforward to
detect their presence, quantity and distribution. It is not possible to
lay down firm rules about the quantity of comment required to describe
adequately a given quantity of code. However, as a rule of thumb, we
might expect that in a body of source code with a healthy pattern of
commenting the distribution of comments between files or functions would
be proportionate to some measure of the quantity of code (possibly
weighted to account for intricacy of decision structure).

3.3.2.3 Questions relating to structuredness

• STRUCT1

Which functions are implemented with more than a single entry or exit
point?

• STRUCT2

How many loop constructs are coded with premature exit behaviour?
• **STRUCT1**
  Which classes or functions contain abnormally complex logic?

• **STRUCT2**
  Which classes or functions contain abnormally lengthy source code?

• **STRUCTS**
  How many test executions of a function are required to provide a particular level of code test coverage?

The practice of structured programming is also widely recommended. There are two related, but distinct, attributes that are presented as alternative models of structuredness:

• the mathematically rigorous model, which sees structuredness as relating to the exclusive use by the programmer of structured language constructs, also known as the Bohm-Jacopini constructs, first described in Bohm and Jacopini (1966); and

• the rather vague proposition that functions should be decomposed into 'manageable' units of decision complexity.

The first two questions relate to the rigorous notion of structuredness, and represent the main deviations from a coding style that requires exclusive use of pure constructs of either sequence, selection or iteration (the Bohm-Jacopini constructs). In C++, the Bohm-Jacopini constructs are typically expressed as follows:

• sequence is expressed in the organization of code into functions containing a mix of decision and non-decision statements;
• selection is expressed using the 'if/else' or 'switch/case/default' structures; and
• iteration is expressed using 'while', 'do/while' and 'for' structures.

The following constructs are generally regarded as not being consistent with structured programming:

• cutting short execution of a function with a 'return' statement anywhere except at the end of the function;
• any unconditional jump (i.e. use of the 'goto' keyword in C++);
• entering an iteration construct at any place except the start (there is no legal way to do this in C++);
• exiting an iteration construct from any point except the start (this can be done with the C++ keyword 'break' used in a loop construct); or
• cutting short a single iteration within an iteration construct (this can be done with the C++ keyword 'continue').

Questions STRUCT3 and STRUCT4 are based on the vague view of structuredness. They represent attempts to set and calibrate rules of thumb for acceptable levels of different kinds of source code content at the class and function level. The use of the adverb 'abnormally' in each of these questions implies that practical conversion of the question into a metric in the next phase of the analysis will involve some kind of threshold at which the attributes concerned are seen as becoming unacceptable.

Question STRUCT5 also relates to this vague view of structuredness, but focuses on the effect of structuredness on the much more concrete notion of testability. As the number of decision points in a function increases, so does the number of separate unit test cases which must be exercised if the developer is to be justifiably confident that all
possible behaviours of the function are appropriate. Broadly, each individual decision point represents a small risk to the project that may be managed by testing. The more decision points, the more testing is required for the development team to discover inappropriate behaviours. If thorough unit testing is not done, the excess decision points increase the risk that the under-tested components will contribute to failures in integration testing or, worse still, in production operation of the system.

3.3.2.4 Questions relating to conciseness

- **CONCISE1**
  Which classes or functions contain an abnormally large amount of source code text?

This question considers the contribution of volume of source code to risks associated with runtime performance, maintainability, and modifiability.

3.3.2.5 Questions relating to legibility

- **LEGIB1**
  How much source code does the programmer need to examine to understand the semantics of a module's interface?

- **LEGIB2**
  How much source code does the programmer need to examine to understand the operation of a module's implementation?

The work practices of the industry team included use of code inspection or review techniques as described in section 2.2. The ease or difficulty with which a component can be read and understood is obviously an important contributing factor to the cost-benefit relationship for code.
reviews, and hence to overall project outcomes. While questions in other sections of this chapter cover the effect of source code bloat in increasing the reading workload on a developer, the current two questions are designed to focus on the increase in cognitive load on the developer triggered by the couplings involved in a component's interface and implementation.

3.3.2.5 Questions relating to augmentability

- AUGMENT1
  How much developer time needs to be expended to accommodate typical changes required to extend the system?

- AUGMENT2
  How much software build and test time needs to be expended to accommodate typical changes required to extend the system?

Both of these questions above are posed in terms of a 'typical' change. It is intended that the kind of change considered typical would be provision of a new variation on existing functionality. For example, a typical change to a personnel system based on a relational database might be the addition of a new data entry screen or report. In the case of the industry team associated with the current project, building a real-time control system, a typical change might be adding support for sending commands to or receiving data from a new class of control device.

One of the major advantages claimed for the object-oriented paradigm is its supposed support for an incremental style of development. Projects based on earlier development techniques would usually have a schedule including a 'big-bang' integration phase just before system testing. High level object-oriented development methodologies such as the Booch
methodology tend to propose an iterative development cycle with continuous integration of new functionality, and periodic releases intended to act as stable benchmarks for future development:

Rather than setting aside a single period of formal integration toward the end of the life cycle, the object-oriented life cycle tends to integrate the parts of its software (and possibly its hardware) at more regular intervals. This practice thus spreads the integration risk more evenly through the life cycle rather than back-loading the development process, where there is less room to maneuver if things go wrong.

(Booch, 1996, p. 75).

Another benefit claimed for object-oriented development is the supposed ease with which previously developed components may be reused to extend the system. Where reusable components are identified, their ease of use can be assessed in terms of the amount of work involved in applying them to implement such an extension. In C++ this attribute might be measured in terms of the number of pure virtual functions which need to be implemented to inherit from an abstract base class. The benefit of reusability can be seen as depending on the ratio of the amount of code involved in fulfilling the requirements of the framework to the amount of code that would be needed to implement the desired extension without reuse.

3.3.2.7 Questions relating to abstractness

- ABSTRACT1

How likely is it to be able to make changes to the implementation of a service offered by a module interface without requiring corresponding changes in the clients who use the service?
ABSTRACT

How easy is it to change the internal representation of a module's state?

These questions are posed to support the measurement of effectiveness of the separation of the interface from the implementation of a component. When a supplier module changes its internal representation or the way it implements one or more of its functions, there is a compatibility issue between source code and products based on the old and new forms of the supplier. Carroll and Ellis (1995, pp. 158-166) define the following hierarchy of compatibility over such a change:

- **Process compatibility**
  New versions of software must be dynamically loaded and become active without stopping the parent process.

- **Run compatibility**
  Executables that implement the client module remain valid but need to be restarted.

- **Link compatibility**
  Executables that implement the client need to be relinked.

- **Source compatibility**
  Modules that implement the client need to be recompiled, but do not require any changes.

The different kinds of compatibility are listed in order of decreasing desirability, with changes which maintain process and binary compatibility causing the least disruption and risk, while link and compile compatibility imply the expenditure of increasing amounts of CPU time to incorporate the change into an integrated system build. Beyond these different kinds of compatibility, there are a number of different
levels of incompatibility, which cannot be remedied without effort by a human developer to change some part of the client module. In order of decreasing desirability, these are:

- **Implementation incompatibility**
  
  The source code which implements the client module requires rework to accommodate the supplier changes.

- **Interface incompatibility**
  
  The source code which defines the interface of the client module requires rework as well as the client's implementation.

- **Contract incompatibility**
  
  The changes to the supplier compromise the ability of the client to fulfill its responsibilities in the overall system design, requiring rework to the design itself and possible reassignment of responsibilities.

The consequences to a client of a change in the representation of, or services provided by, a supplier depend (evidently) on the nature of the change itself, and also on the nature of the relationship. In C++, where the typical module is a class, the relationships supported can be categorized as follows:

- **inheritance**
  
  The client module is a subclass of the supplier.

- **containment**
  
  The client module has an instance of the supplier as part of its state.

- **parameter passing**
  
  The client module receives an instance of the supplier module as a formal parameter to one or more member functions.
• global access
  One or more member functions of the client module access global instances or static member functions of the supplier module.

• local instantiation
  One or more member functions of the client module create local instance variables of the supplier module.

Of the relationships listed, inheritance, containment and parameter passing relationships are expressed in the source code of the interface of a class and so give rise to the greatest risk of change propagation.

All of these kinds of relationships can be categorized according to the most visible part of the interface where the relationship is referenced. This categorization is of value as the access control rules of the language help to protect client classes from source code incompatibility by preventing the client source code mentioning methods or data defined in the restricted parts of the supplier class definition.

The access control rules of C++ divide the member data and functions of a class into three groups:

• the private part, which is only accessible to the class itself and other classes and functions which are declared to have a friendship relationship with the class;

• the protected part which is accessible to the class itself and also to its subclasses; and

• the public part which is available to any client.
Comparable object-oriented languages tend to have similar access control rules. In particular, the definition of separate private and public interfaces to modules is an expression of fundamental concepts of the object-oriented paradigm. The protected access control category was introduced into C++ as a result of the observation by the language's designers that programmers in early versions of the language frequently used the C++ 'friend' keyword to grant access to the private interface of a class to its subclasses (Stroustrup, 1994, p. 301). This usage suggested that clients by inheritance frequently required access to a wider interface than other clients of a supplier class.

Not all languages used for object-oriented programming support the notion of a protected part of the interface. There is no reason why other languages might not provide further access control categories beyond the three provided in C++. For example, Java supports a fourth level of protection, sometimes called 'friendly', which is the default access control policy applied when none of the access control keywords (private, protected or public) are specified. Methods and attributes which have this protection level are directly accessible from code defined in the same package as the supplier, but not from outside the package.

Given that we expect object-oriented programming languages to provide a syntax for defining one or more restricted interfaces for different categories of client, the content and extent of the different interfaces defined for a module have an evident impact on the risk of change propagation from supplier to client. In a system with N datatypes, the number of potential supplier-client interfaces is equal to $N(N-1)/2$. 
Each such relationship could belong to any of the following categories:

- **no interface at all**
  The potential client has no coupling at all with the potential supplier.

- **opaque interface**
  The potential client is aware of the supplier module's existence and is capable of remembering the identity of an instance of the supplier, but has no knowledge of the supplier's behaviour or representation.

- **abstract interface**
  The potential client is aware of some group of operations which the supplier supports but has no knowledge of its representation.

- **partial concrete interface**
  The potential client knows about some part of the internal representation of the supplier (and may also know of some of the supplier's operations).

- **complete knowledge**
  The potential client knows the complete internal representation of the supplier (knowledge of the supplier's operations is now irrelevant as the supplier no longer has any secrets and the client can directly apply any transformation it wishes to the supplier's state).

These categories are listed in order of likely increasing risk with regard to change propagation. We can categorize the kinds of changes that a supplier module might undergo as follows:

- **implementation**
  A change is made to the internal representation of the type or to one
or more of the algorithms used to implement its operations.

- **interface**
  
  A change is made to some part of the type's externally visible interface (this implies that some part of the implementation will change).

- **contract**
  
  There is a change to the high-level responsibilities the type provides to the system (this implies that the interface and implementation will both change).

We can tabulate the range of propagation consequences for each of these kinds of change for potential clients according to this client interface taxonomy:
Table 2: Effect of abstraction on change propagation

<table>
<thead>
<tr>
<th>Client interface</th>
<th>Implementation change</th>
<th>Interface change</th>
<th>Contract change</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>Change cannot possibly propagate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Opaque</td>
<td>Change cannot possibly propagate</td>
<td></td>
<td>Change may propagate</td>
</tr>
<tr>
<td>Abstract</td>
<td>Change cannot possibly propagate</td>
<td>Change may propagate to client’s implementation, unlikely to affect client’s interface</td>
<td></td>
</tr>
<tr>
<td>Partial concrete</td>
<td>Change may propagate, may affect client’s interface, implementation and contract</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The most desirable outcomes, the restriction of the change to the supplier, are found in the top left corner of the table. As we move downwards and to the right, consequences become more and more serious.

It is possible for a supplier to have multiple interfaces of the abstract or partial concrete categories with different subsets of the supplier’s operations and data members visible.

Under the categorization of Table 2, the public and protected interfaces of a C++ class might be either abstract or partial concrete interfaces according to whether any data members are exposed in the particular interface.

In the specific case of C++, excluding client classes that have the privileged access to the supplier granted by the 'friend' keyword, we would expect:
that changes which are confined to the private part of a class's interface would be at least compile compatible with all clients; and

that those which are confined to the protected part would be compile compatible with clients by relationships other than inheritance.

Another useful categorization which applies to containment and parameter passing relationships (but not to inheritance, in C++ at least) is between relationships where the supplier is specified by value versus those where the supplier is specified by reference (i.e., using either the C++ pointer or reference syntax). While this may seem a minor distinction from the developer's point of view, containment or passing by value both require the compiler to see the definition of the supplier class before the client's for the client to be compilable. This has a number of implications:

- In the case of a containment relationship, a change to the internal representation of the supplier class is implicitly a change to the internal representation of the client.

- In the case of a parameter passing relationship, every invocation of the function which receives or returns the client causes implicit execution of the assignment operator or copy or conversion constructors of the supplier class.

- Code in which two classes both have inheritance, containment or parameter passing relationships by value to the other is absolutely uncompileable, as neither definition is compilable before the other.

Because of the first two points above, a change to the definition of a supplier mentioned by value can not be expected to have compatibility better than compile compatibility with respect to any component which
includes the client's interface definition. This promise is true whether or not the implementation part of the component directly refers to the supplier, thus containment and passing by value are major contributors to the risk of cascading recompilation requirements. If the same relationships were established by reference (and no relationships by value with the same class existed), the definition of the client would be made compilable by a simple forward declaration of the supplier class. The implementation of the client would still be likely to require inclusion of the supplier's definition, and would thus be restricted to compile compatibility with respect to supplier changes. Other components which include the client definition would now be likely to be link compatible.

While use of indirection in containment and parameter passing relationships has definite advantages in terms of cascade of change requirements from a supplier to its immediate and indirect clients, the use of indirection is not without some dangers. Firstly, the elimination of the implicit copy operation means that the implementation of the client operation has access to the same instance of the supplier type that the calling context sees, and the operation becomes capable of causing unexpected side effects by modifying the passed object. Secondly, particularly when the supplier class is the return type of an operation, return by reference implies that the return value must be an object which exists independently of the execution context of the operation, i.e. a class member variable, global variable or a static local variable. The fact that the operation needs to use a location which has a lifetime longer than the current invocation raises the possibility of side-effects, and may also prevent the operation being re-entrant or thread safe.
For the latter reason some relationships need to be specified by value, which interacts with the third point raised above to create a danger of contract incompatibility. The existence of a relationship between two classes by value in one direction excludes the possibility of a relationship by value in the opposite direction, either directly or indirectly through a cycle of relationships of any length. One of the advantages proposed for the object-oriented paradigm is the ability to support iterative development. A requirement for an operation which needs to return a supplier class by value discovered late in the evolution can cause just such a cycle and may necessitate substantial rework to the architecture of the system.

3.3.2.8 Questions related to reusability

- **REUSE1**
  How many other modules reuse the current module by inheritance?

- **REUSE2**
  How many modules reuse the current module by means other than inheritance?

Another frequently referred-to benefit of object-oriented style is its support for re-use of software components within and between projects. Re-usability is seen as a quality goal for two reasons:

- Components that have been created and used previously may have been exercised and tested in their previous instances of use to an extent where many potential problems with them have been discovered and fixed (whether this benefit is real or not depends on the history of the component).
• Components which lack the attribute of re-usability are also likely to be fairly hard to subject to rigorous unit testing.

The two questions above really measure the extent of re-use rather than re-usability itself, but this may be a reasonable indirect indicator. Some of the early questions relating to attributes like self-containedness and augmentability also have an effect in re-use, as the absence of either of these desirable attributes will make re-use harder to achieve.

3.3.2.9 Questions related to buildability

- **BUILD1**
  To how many other modules' interface does the current module's interface have dependencies?

- **BUILD2**
  To how many other modules' existence does the current module's interface have a dependency?

- **BUILD3**
  How many other modules' interfaces have a dependency on the current module's interface?

These questions are designed to characterize the cascading effect of recompilation requirements spread by inter-module dependency.

As noted in section 3.3.2.7, the (directional) relationship between two modules in a system can be categorized according to how much knowledge each module has of the other. The more knowledge shared, the more likely that a change in one module will invalidate the object code built from the other module (assuming separate compilation of the modules). The magnitude of this effect will depend to some extent on choices made by
the language designers and implementors. For example, the Java language uses implicit indirection for most relationships, which reduces the effect of representation changes at least. C++, with its traditional emphasis on minimizing unavoidable drains on run-time performance, does not employ this kind of technique, and cascading recompilation requirements are a common experience of developers using the language for large scale projects. Thus, buildability is again closely dependent on earlier mentioned attributes like self-containedness and abstractness.

3.3.3 Metric selection

While the questions laid out in the preceding sections were derived from the distinct base quality attributes listed, it can be seen that there are a number of recurring issues, including:

- the volume of writing/reading/thinking a programmer has to do to create, extend or exploit project source code;
- the number and level of dependencies between modules; and
- the separation between a module's interface and its implementation.

A number of metrics were chosen for implementation based on the goals and questions described. The metrics implemented were grouped into three categories. The categories chosen were intended to assist in presenting the metrics as illustrations of different coherent views of the project's source code. The three groups of metrics are listed below, together with a brief description of the view of the source code which each is intended to support.

- Procedural metrics

  This group of metrics is organized around a view of the software in
which the individual procedure or subprogram is the most significant unit. This is quite a low-level view of source code, emphasizing the amount of software that has been written and also its organization at the subprogram level of detail.

• Structural metrics
This group of metrics is oriented to a view of software as a complex of interacting modules. The focus in this group of metrics is very much on the network of relationships between the modules, rather than the modules themselves.

This view represents a higher level conception of a software system than the procedural view. The focus on relationships means that the measures determined in this view can be seen as attributes of the system as a whole rather than of the system's individual components.

• Object-oriented metrics
This application of the object-oriented paradigm to software design and construction is an issue of wide concern at the present time. Chidamber and Kemerer (1991, 1994) have proposed a suite of measures as an initial basis for measurement of object-oriented projects.

This group of metrics was included in recognition of the present interest in object-oriented programming, and of the fact that the Chidamber and Kemerer suite of metrics is the most widely discussed set of metrics specifically related to the object-oriented paradigm in the literature to date. These metrics adopt a higher level view of a software system than the procedural metrics, but are focused on the modules (classes) of the system themselves, rather than their relationships, so they can be seen as being less holistic than the structural measures.
Each of these groups is discussed in more detail in subsections 3.3.3.1 to 3.3.3.3. The metrics in each group are categorized according to whether they are obtained directly by analysis of the source code (i.e., usually some kind of count), or by applying a formula to other metrics. We will use the term 'primary metric' to denote metrics which are obtained directly from the source code and 'derived metric' to denote metrics which are derived by applying some kind of formula to the outcome of other metrics. For each of the metrics discussed, an abbreviated name or 'tag' is presented, together with an informal description of the basis of calculation of the metric. The exact algorithms used for the calculation of the different metrics in the implementation of the analyzer are discussed in the following chapter.

3.3.3.1 Procedural metrics

The procedural metrics group includes the following primary metrics:

- LOC
  Lines of code;

- NVC
  McCabe's cyclomatic number; and

- COM
  Lines of comments.

In addition to these, the following derived metrics are calculated:

- L_C
  Lines of code per line of comment; and

- M_C
  McCabe's cyclomatic number per line of comment.
Each of these measures can be reported at three levels of detail: at the level of the individual function, at the level of the class, and for the system as a whole (the analyzer views the system as the set of source code submitted for a single run).

These metrics are intended to provide answers to the questions in the areas of self-descriptiveness, legibility, conciseness and structuredness. The McCabe cyclomatic number also has a particularly direct relationship to the testability of a component, although testability was not identified as one of the quality model attributes to track (in retrospect, this omission may well have been a mistake).

Note that the two derived metrics are ratios. In the terminology of physics these are intensive properties rather than extensive properties, i.e. they are expected to vary within the same range whether measured over small or large quantities of the test material (in this case source code). Intensive properties are particularly useful when trying to establish numeric standards for quality, because it is possible to lay down criteria that do not depend on estimation of the size of the product. It is not uncommon to find software development organizations with informal or formal guidelines laying down the expected ratio of comment to code. These guidelines are often couched in the form of what we are referring to here as L_C (i.e. LOC/COM). We are also presenting M_C as a similar ratio which offers a weighting more suited to checking that the comments are distributed proportionately to the logical complexity of the source code, rather than its textual length.
1.3.3.2 Structural metrics

The structural metrics group is built upon foundations suggested by Henry and Kafura (1981, 1984), as part of their information flow measurement system, with extensions which are intended to highlight particular questions relating to the application of object-oriented technology to development.

The basic components of this group include the following primary metrics:

- **FI**
  
  Fan-in; and

- **FO**
  
  Fan-out.

Fan-in and fan-out are discussed in a number of places as metrics of a module of a software system. Henry and Kafura adopt definitions of these metrics which focus on the flow of information between modules. The fan-in of a module is the number of other modules which are capable of causing information to flow into a module. For example, the fan-in count would include external modules which perform direct write access to the reference module's data; external modules which invoke operations belonging to the reference module which cause its state to change; and external modules which the reference module invokes which return data to the reference module. Similarly, the fan-out is the count of other modules which are capable of causing information to flow out of the reference module. Examples of this are external modules that read the reference module's data; external modules upon which the reference module invokes state-changing operations; and external modules that invoke operations of the reference module which return results.
In addition to these primary metrics, the Henry and Kafura framework proposes a single derived metric, which is called IF4. IF4 is calculated according to the formula:

\[ IF4 = ( FI \times FO )^2 \]

Where \( FI \) and \( FO \) signify fan-in and fan-out as defined above.

\( FI \) and \( FO \) are both self-evidently measurements of coupling between modules, differing only in the direction of the dependency relationship. The justification for multiplying and then squaring them tends to be described in terms of a conscious attempt to develop a metric that favours particular patterns of coupling. Shepperd and Ince (1993, pp. 41-50), present a detailed criticism of the various formulations of the IF4 measure by its original proponents.

Despite raising questions about the theoretical basis of the measure, Shepperd and Ince report a range of empirical studies in which it has been found to be of possible value in prediction of maintenance trouble spots within large systems.

The information flow complex of metrics predates the current wave of object-oriented tools, techniques and languages, but the view of architectural quality of software is at least consistent with some of the proposed benefits of those tools. It was decided to supplement the 'classical' \( FI \), \( FO \) and \( IF4 \) measures with two sets of parallel measures intended to emphasize the use of different aspects of the object-oriented paradigm to reduce the architectural risks inherent in large systems.
discussed in section 3.3.2.7, C++ offers two important ways of reducing coupling via relationships:

• the use of indirection in containment and parameter relationships reduces the flow of knowledge between suppliers and first and n-th order clients through the compilation process; and

• the use of access control specifiers reduces the flow of this knowledge through the source code itself.

For each of these two kinds of language supplied constraints, a parallel set of the information flow measures was derived giving us:

• $F_{iv}$, $F_{ov}$, $IF_{4v}$
  the fan-in, fan-out and IF4 of a module taking into account only relationships in parts of the module which are externally visible (public or protected parts of a C++ class definition); and

• $F_{ic}$, $F_{oc}$, $IF_{4c}$
  the fan-in, fan-out and IF4 of a module taking into account only relationships which create a dependency requiring the supplier's definition to be seen by the compiler before the client's definition can be compiled (the letter 'c' is a mnemonic for the property of 'concreteness' which can be seen as the opposite of the abstractness of the relationships which are filtered out of this view).

This group of metrics, as augmented, is intended to answer questions relating to self-containedness, augmentability, reusability, and buildability. The metrics are reported at the level of the module and the system.
While both of the modified measure sets relate to quantification of the risks associated with inter-module coupling, they focus on different aspects of these risks. From the observations in section 3.3.2.7, it is evident that not all relationships create the same degree of risk.

Relationships which are concealed from clients by the access control mechanisms of the language are unlikely to require code rework in those clients (although the change may imply that the clients have to be recompiled). Thus, the modified measures with respect to visibility are intended to relate to the danger of change propagation in the forms of interface, implementation or contract incompatibility.

Similarly, relationships which can be characterized (in the source code of the client interface at least) as opaque are likely to survive changes to the client representation without requiring widespread recompilation, i.e. direct and indirect client modules are likely to be link compatible. The modified set of structural measures filtered on concreteness emphasizes the risk to project success posed by relationships which are not opaque, and hence are likely to cause supplier changes to require widespread recompilation of direct and indirect client modules.

3.3.3.3 Metrics suite for object-oriented design

This group of metrics consists of the following metrics:

- WMC
  Weighted methods per class;
- DIT
  Depth of inheritance tree;
• NOC
  Number of children;

• CBO
  Coupling between objects; and

• RPC
  Response for class.

These metrics have been proposed by Chidamber and Kemerer (1991, 1994) primarily as a basis for measuring the scale of object-oriented projects, much as LOC and MVG are seen as appropriate tools for different views of the scale of software built under earlier paradigms. This group of metrics for implementation was selected because the work of these authors represented the most widely discussed group of source code metrics relating specifically to object-oriented paradigm at the time of the project. It was realized that data being gathered to report the other metrics selected would enable the calculation of four of the five metrics in the group, hence these metrics were implemented. No specific relationships were identified between the metrics in this group and the quality model attributes.

3.3.3.4 Summary of the relationships between selected metrics and the quality model attributes

The following table summarizes the relationship between the quality model attributes and the individual metrics selected for implementation.
Table 3: Relationships between Metrics and Quality Model Attributes

<table>
<thead>
<tr>
<th></th>
<th>self-containedness</th>
<th>self-descriptiveness</th>
<th>structuredness</th>
<th>legibility</th>
<th>argumentability</th>
<th>abstractness</th>
<th>reliability</th>
<th>buildability</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Non-comment, non-blank lines of code</td>
</tr>
<tr>
<td>LOCm</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LOC calculated on a per-module basis</td>
</tr>
<tr>
<td>LOCf</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LOC calculated on a per-function basis</td>
</tr>
<tr>
<td>MVG</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>McCabe's cyclomatic number</td>
</tr>
<tr>
<td>MVGm</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MVG calculated on a per-module basis</td>
</tr>
<tr>
<td>MVGf</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MVG calculated on a per-function basis</td>
</tr>
<tr>
<td>COM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lines of comment</td>
</tr>
<tr>
<td>L_C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ratio of LOC to COM</td>
</tr>
<tr>
<td>M_G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Ratio of MVG to COM</td>
</tr>
<tr>
<td>FI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>Fan-in</td>
</tr>
<tr>
<td>FIv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>Fan-in (visible relationships only)</td>
</tr>
<tr>
<td>FIr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>Fan-in (concrete relationships only)</td>
</tr>
<tr>
<td>FO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>Fan-out</td>
</tr>
<tr>
<td>FOv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>Fan-out (visible relationships only)</td>
</tr>
<tr>
<td>FOr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>Fan-out (concrete relationships only)</td>
</tr>
<tr>
<td>IF4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>Information Flow</td>
</tr>
<tr>
<td>IF4v</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>Information Flow (visible relationships only)</td>
</tr>
<tr>
<td>IF4c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>Information Flow (concrete relationships only)</td>
</tr>
</tbody>
</table>

3.4 Theoretical bases of validation

Before performing theoretical validation, it is necessary to define a framework within which that validation can take place. In particular, we need to arrive at an operational definition of validity in this context. It would be convenient to know an algorithm that can be applied to the definition of each measure, the outcome of which was a single figure of merit relating the metric with its value in the software engineering process. Sadly, the value with which we are trying to correlate is
itself intangible, and it is difficult to conceive a meaning for any such numerical scale.

An alternative strategy is to validate by proving a link between a measure and attributes which demonstrate the application of some form of approved technique, which we accept on trust (for the purposes of the current discussion) is expected to improve practical outcomes. The sorts of techniques of which we are likely to approve will of course be drawn from the mainstream of the software engineering discipline and might include such items as:

- modularization of code
- low inter-module coupling
- high intra-module cohesion
- restrictions on the length/complexity of functions
- minimization of violations of the strong typing system
- application of structured design/programming
- application of object-oriented programming/design
- appropriate use of design patterns
- appropriate use of modern language technology including:
  - separate compilation;
  - information hiding;
  - access control; and
  - inheritance.

The list above is not intended to be exhaustive, but it does illustrate a range of techniques which intuitively might be susceptible to some form of indirect or direct measurement. The list is not restricted to techniques for which the current project implemented measures, so some items above will not be discussed further.
The consideration of validity of selected metrics is being performed as a prelude to implementation of a tool to capture those metrics. At this point, the metrics we are considering exist as only informal descriptions in the literature, and (in the case of the novel variations proposed on the information flow metric) in section 3.3.3.2 of the current thesis. Ideally, we would proceed to formal unambiguous definitions of the metrics before performing any validation. In the context of the current project, it was recognized that the implementation phase was likely to involve pragmatic decisions to define the final form of metrics in such a way as to make data capture easier (or indeed feasible). For this reason, a precise definition of the calculation methods used to obtain the different metrics is deferred to the next chapter, which discusses the practicalities of implementing an analyzer program. Some of the sections in the remainder of the current chapter do, however, foreshadow certain of these decisions by talking about the degrees of freedom available in the implementation of the imprecisely described metrics on our agenda.

3.4.1 Observations from measurement theory

There is a theoretical discipline of measurement that offers a number of theoretical and notational tools for the discussion of the validity of software metrics. Fenton (1991, pp. 29-33) describes a categorization of measurement scales:

- A nominal scale is one that allocates each measured entity to one of a number of disjoint sets. The sets may be identified by numbers, but do not imply an ordering with regard to the measured attribute.
- An ordinal scale is one that allocates the measured entities among a set of categories that have some inherent ordering relationship.
• An interval scale is defined as a scale that, like an ordinal scale, enables ordering of items. Unlike an ordinal scale, it provides useful information on the magnitude of difference between items at different points on the scale.

• A ratio scale is defined as a scale that, like an interval scale, provides ordering and magnitudes of difference, but which also supports meaningful operations of multiplication and division.

• An absolute scale is a scale that supports ordering, magnitude of difference and multiplicative operations. In addition, an absolute scale is required to be unique for the attribute it measures (i.e. the only permissible transformation between two absolute scales for the same attribute is identity).

It is fairly evident that each of the sets of ordinal, interval, ratio and absolute scales are each successively superset of the preceding sets of scales.

For example, in C++ we might categorize member functions of classes according to their access control (public, protected, private). Although the three categories here are given names rather than numbers, as a measure of the useful attribute of 'hiddenness' we can define an ordering on the three categories: private functions are more hidden than protected functions which are in turn more hidden than public functions.

A set of examples to illustrate interval and ratio scales would be different scales of temperature measurement, Fahrenheit, Celsius and Kelvin. All of these scales meet the requirement for interval scales: we can meaningfully make statements like "Perth will be 10 degrees Fahrenheit warmer than Brisbane today", or "The temperature range between
the freezing and boiling point of water is 100 degrees Celsius. The Kelvin scale is the only one of the three which meets the requirements for a ratio scale. Because the zero point of the Kelvin scale is defined as a point at which the effects associated with heat are absent from a material, it becomes meaningful to make statements like "The boiling point of oxygen is 10 times that of helium". While neither addition nor division are operations which can appropriately be applied individually, to an interval scale observation, the combination of these two operations into the calculation of an arithmetic mean will be meaningful.

While the Kelvin scale is a ratio scale for temperature, it is not an absolute scale as there are other scales that can equally meaningfully be used. A scale that had the same zero point but a unit of measurement larger by a factor of 10 would be equally meaningful. In practice absolute scales are mainly observed where the attribute under consideration is a count of the population of some intuitively defined observable entity. As an example, the number of children might be an absolute scale relating to class sizes in a school.

The classification of scales in this way is useful because it helps us see what kinds of measurement techniques are likely to give us meaningful results. In general, we are interested in processing large volumes of source code, and we will probably want to restrict ourselves to metrics which can be defended as being at least interval scales. This is because the lesser scales do not support the kind of operations we need to combine large numbers of individual observations into summary results by extracting a mean.
3.4.2 Weyuker's properties of software complexity measures

A widely cited axiomatic framework for the evaluation of measures of software source code complexity is the work of Elaine Weyuker (Weyuker, 1988). Weyuker's article proposes a set of conditions which are necessary (but not necessarily sufficient) for a measure to be useful in the assessment of software complexity. In Weyuker's notation, letters like P, Q, R represent different programs, and the magnitude of the measure applied to each program is signified using the modulus notation |P|, |Q|, |R|. The properties suggested are as follows:

- **Property 1:** There exist programs P and Q such that |P| is not equal to |Q|.

For a measure to have any value at all, it has to enable some discrimination between different subjects. This property requires that there be at least some variation in outcome when the measure is applied to different programs.

- **Property 2:** Let c be a non-negative number, then there are only finitely many programs of complexity c.

This property is posed to imply a requirement that changing a program must ultimately cause a change to its measurement. The counterexample supplied by Weyuker to illustrate its value is the cyclomatic complexity metric: as this measure is only sensitive to the number of decision points, it follows that it is possible to add arbitrarily many non-decision statements to a program without changing the cyclomatic number.

- **Property 3:** There exist distinct programs P and Q for which |P| is equal to |Q|.

This property excludes from the world of useful measures metrics which are incapable of returning identical values for distinct programs.
The argument for doing this is that in any selected programming language it is certain that there are dimensions of choice that would allow the definition of programs which are distinct in source code but identical in effect. An obvious example of this would be two programs that differ only in the spelling of identifiers. The requirement for this property is an assertion that programs that are effectively identical should not be ranked differently by the measure.

- Property 4: There exist programs P and Q such that the external effect of P and Q are identical, but |P| is not equal to |Q|.

This property asserts that the measure must apply to the implementation of the effect of the programs and must not be completely determined by the external effect. The importance of this property is that, if it were not adhered to, the measure would provide no discrimination between any of the infinite numbers of possible programs having a given specified external effect, and would hence be of little use in ranking them according to their inherent complexity.

- Property 5: For all programs P and Q, considering also the program P;Q obtained by combining P and Q, |P| + |Q| is less than or equal to |P;Q|.

The justification for this property is the notion that interaction between parts of a program may introduce complexity additional to that present in the components themselves. The amount of added complexity may be zero, but may never be negative.

- Property 6: There exist programs P, Q and R such that the |P| is equal to |Q| but |P;R| is not equal to |Q;R| (in Weyuker's formulation of this property, it is stated twice with the order of composition reversed in the second variant).

Again this property emphasizes the potential of composition to create
complexity additional to that inherent in the components, and asserts that this interaction complexity is not completely determined by either of the interacting components.

- **Property 7:** There exist programs $P$ and $Q$ which are composed of the same statements in a permuted order for which $|P|$ is not equal to $|Q|$. This property emphasizes that the order of statements affects the accumulation of complexity. The example Weyuker gives of this is the effect of nested control structures: she argues that a pair of loop statements may well be seen as having higher complexity when the one is nested within the other than when they are executed consecutively.

- **Property 8:** If two programs $P$ and $Q$ differ only in the choice of names for different elements, then $|P|$ is equal to $|Q|$. It is intuitively obvious that two programs that differ only in choice of names should yield identical measures. Weyuker does suggest a counter example to this property for the case of a measure which is expected to capture the difficulty of understanding of a program: in this case the meaningfulness of names chosen might appropriately cause two such programs to be ranked as more and less understandable.

- **Property 9:** there exist programs $P$ and $Q$ for which $|P|+|Q|$ is less than $|P;Q|$. This is a stronger form of property 5, in which the observation that interaction between parts may contribute a non-negative increment to the complexity becomes a requirement that, for some combinations at least, this increment is strictly positive.

Weyuker's properties have been criticized as being over-restrictive, for example by Shepperd and Ince (1993, pp. 69-70). Toward the end of the chapter, they observe:
This section on model evaluation has described what must be construed as almost a wish list. It is concluded that evaluating software models is altogether more difficult than might be supposed. Consequently, it is more appropriate to address smaller and more manageable problems. The search for 'Holy Grail'-type metrics would not seem to be either productive or feasible at the present time. (ibid., p. 72).

3.4.3 Applying Weyuker's complexity axioms over an extended model of metric use

There is an assumption, which is implicit in Weyuker's work and in most of the discussion of it in the literature, that the outcome of a single run of a measurement tool over a program is a single number representing some view of the complexity of the whole system. It has been noted that none of the classical metrics evaluated using Weyuker's framework is able to demonstrate all of the properties she identifies, and that other writers have criticized the framework as being either too restrictive, or too ambitious, or both.

It is worth noting that the assumption of a single global 'figure of merit' output for a system is not consistent with many of the practical proposals for software measurement. In these it is much more common to see whatever measurement technique is being illustrated being applied to a set of programs or modules, and much of the meaning imputed to the measurement being closely related to the ranking of an individual module's outcome against those of its peers. McCabe discusses the use of his cyclomatic complexity technique to establish a standard of subdividing modules which reach complexity 10 (1976, p. 314). Another example is the work of Shepperd and Ince (1989), who discuss use of Henry
and Kafura's information flow measure in an iterative process of refinement of design where the modules which give rise to outlying measurement outcomes are given special consideration for rework.

The consideration of Weyuker's properties as applied to single scalar measurements is intuitive, and follows naturally from the use in the axioms of the framework of the operations of equality, inequality and magnitude comparison, which we naturally associate with scalar number systems. Providing we are able to define meaningful semantics for these four operations, we can consider the application of the properties to measurement outcomes more complex than single points on one or other number scale.

As an illustration, let us assume that the outcome of some defined measurement process on a program $P$ is a collection of elements where each element consists of the four components: tag, scope, measure, and context. Each element represents a single micro-level observation about the program $P$ or some part of it, with the components having the following meanings:

- the tag identifies the particular metric of which the current observation is an instance;
- the scope identifies the part of program $P$ over which the current observation is calculated;
- the measure is a scalar numeric value which is the outcome of the micro-measurement specified by the tag and the scope; and
- the context is any additional interpretive information about the current observation (e.g. ranking of this observation among comparable ones in the same response set, or against standard thresholds).
There is no requirement that a set of such elements be homogenous in any particular way; in particular it would be conceivable to use such a set to represent the outcome of a process which measured more than one metric (e.g. LOC and COM). The scope of individual observations need not all be at the same level of granularity; the set might include summary observations at one or more levels of detail (i.e. function, source file, module, project), and these observations need not be independent. Using Weyuker's notation for the complexity of a program $P$ being $|P|$ we would say that

$$|P| = \{ O_1, O_2, \ldots, O_n \} ;$$

where each $O$ represents an observation tuple with the structure:

$$O_i = ( \text{tag, scope, measure, context} ) .$$

Note that set notation has been used for $P$, hence the positions and indices of the members are not a significant part of the representation.

The structure described allows us to express a very wide range of measurement outcomes: to apply Weyuker's axioms we must use this notation to describe the outcomes of particular sequences of measurement operations, and define operations which enable whole response sets to be compared.

Equality and inequality operations are intuitively easy to define and express. One obvious condition for equality of two response sets $|P|$ and $|Q|$ is that for each observation $O_i$ in $|P|$ there is an identical observation $O_j$ in $|Q|$ and vice versa (note that the indices are not significant). The inequality operation is obviously the negation of the equality one. Other equality operations are possible, perhaps based on
summation or averaging of observations. Our reason for selecting memberwise comparison for the equality operation is that it maximizes the sensitivity of the comparison to meaningful differences between a pair of response sets. As we have observed earlier in the current section, ordering of items within a response set is not considered meaningful, and two response sets consisting of the same observations in a different order would be considered identical. No summation or averaging operation which could be applied across two such sets could yield different values, although it is easy to see that response sets which are not identical under the memberwise comparison operation could yield identical sums or averages. This implies that the memberwise comparison policy gives us an operation which makes it easier to make distinctions between different response sets. We shall see in section 3.4.5 that this increased ability to discriminate makes it considerably easier to meet the requirements posed by Weyuker for a useful measurement.

Definitions for operations comparing magnitude are less self-evident, indeed there are several candidate definitions amongst which no single one can be seen as clearly appropriate in all cases. Fortunately, we do not need to prove that our chosen definitions for these operations are uniquely correct, only that they help to convey useful information. Hence, the different possibilities can be allowed to coexist, so long as in any particular analysis we fix on a single definition and justify its selection.

The intuitive candidates for comparison operations are based on selecting or deriving a single scalar datapoint representative of each set, then applying normal scalar comparison operations to these representatives.
Possible strategies for this would be extraction of a sum, arithmetic or geometric mean, or median from the set of observations at a particular level of detail. These strategies have the advantage of yielding comparison operations which behave similarly to normal scalar comparisons.

An alternative approach would be for each comparison operation to select the most extreme reading from each set in the direction of the comparison (e.g. the greater than comparison of two sets would be performed by comparing their largest elements, the less than comparison by comparing their smallest elements). The comparison operations generated by this policy have one particular non-intuitive feature: for two complexity responses \(|P|\) and \(|Q|\) it is possible for \(|P| < |Q|\) and \(|P| > |Q|\) to be true at the same time. For an example of how this may be true consider a pair of response sets \(P\) and \(Q\), for which the measure parts of the observations are respectively \(\{1, 2, 15\}\) and \(\{3, 11, 9, 7\}\). Under the policy of comparison by extremes, we find that both \(|P| < |Q|\) and \(|P| > |Q|\), in defiance of the normal expectations of these operations.

The policy of comparison by extremities may seem contrived, but does serve to emphasize the outlier datapoints in each set, which are arguably of more significance in terms of guidance for future action than any knowledge about the general tendency of the 'typical' datapoints. An approach to the use of metrics which places particular emphasis on such outlier values has been described by Shepperd and Ince (1989).

3.4.4 A data model for measurement of object-oriented software

The preceding section has introduced the concept of a response set as being the outcome of applying a measurement tool to a body of software
source code. The response set notation has deliberately been specified in such a way as to be able to express the outcomes of a wide range of measurement applications. The description of a response set in set theory terms gives us flexibility and expressive power on the theoretical level, but order-insensitivity of the response set notation is inconvenient for most practical purposes. For this reason we would wish to select a canonical way of selecting a standard order for expression of any particular response set. Examining the data held in each individual observation, it is clear that, of the four elements that make up an observation, the tag and the scope are best seen as input variables, while the measure and the context are outcomes. This leads us to suggest a canonical format for external representation of response sets as a (potentially) two-dimensional associative array, with the tag and the scope of a particular measurement as keys, and each cell yielding a binary tuple containing the scalar measure and its context information.

Having identified the tag and the scope as the index types of the representation, it is important to be able to define the range of values each will adopt in response sets generated by a particular measurement tool.

The range of the tag is simple to predict: for each distinct measurement algorithm applied by the measurement tool, a tag will be selected at the time the algorithm is implemented, and this tag will be attached to all observations representing the algorithm.

The range of the scope index is also tied up with the way that the measurement algorithms are defined. The scope of a particular
observation is informally defined as the part of the program to which a particular measurement relates (a formal rigorous definition would need to take into account the nature of the particular measurement algorithm the scope was being applied against).

There is nothing in the response set notation that restricts us to using similar scopes for different measurements. Neither is there anything that prevents us from choosing to do so. On balance, it is convenient to implement the analysis tool using a consistent model of scope across the different measurements that will be reported. The ways that different kinds of scope were defined and measurements across them were implemented were treated as implementation decisions, which are described in the chapter following this one. A brief preview is presented directly below in order to provide an example of the sort of information we expect to see in the scope part of the response set.

The scope model chosen for implementation in this project consisted of the following layers:

- **Extent**
  A localised piece of code in a contiguous section of a single source file which is identified as containing some measurable phenomenon;

- **Function**
  A scope which contains one or more extents which is identified as being a single subprogram in the analyzed language (typically the multiple extents associated with a function will be zero or more declarations and zero or one definitions of the function);

- **Module**
  A scope which contains a group of functions and other extents which
are bound together using the syntax of the analyzed language for encapsulation (e.g. a class, namespace or struct in C++, a class, interface or package in Java, a class or package in Ada);

- Project
  A scope which contains all of the modules, functions and other extents which are analyzed in a single run of the analysis tool.

At the lowest level, all of the raw information which feeds the algorithms of the analysis tool will come from individual extents, however an important part of the role of the tool is to organize the raw information into a more useful format. Thus, the tool would be expected to build up a database of observations at the extent level, allocate each extent to one or more of the higher level scopes, and generate a response set which summarizes the observations at one or more scope level. The observations in a response set generated in this way may be independent, or the analyzer may be set up in such a way as to return a response set that contains redundant information. For example, observations for higher level scopes can be deduced by summation of observations for lower level scopes. Whether or not the analyzer returns redundant information of this form is an implementation decision and is irrelevant to the theoretical considerations discussed in the next section. We shall assume that on receipt of a response set we may obtain any form of summation or averaging of the independent underlying observations within it. This may be done by extracting summary observations from the response set itself or by further calculations on the independent observations.
3.4.5 Evaluation of selected metrics using Weyuker’s framework and the extended model

In this section, the effect of the response set notation in the application of Weyuker’s axioms will be examined, using examples drawn from procedural and structural metrics selected by the CQM analysis presented earlier in this chapter. The response set notation is intended to model the return to the experimenter of a richer set of data than the single scalar values envisioned in Weyuker’s original work, however a single scalar value can obviously be represented by a response set containing a single item. Thus, the focus of the analysis will be on the ability or otherwise of a metric tool returning a response set to satisfy Weyuker’s properties in cases where a scalar result can not be demonstrated to do so.

3.4.5.1 Procedural metrics (LOC, NVC and COM)

Of this group of metrics, it is appropriate to exclude the COM (number of comment lines) metric, on the basis that it is not intended for use as a metric of complexity. The other two are widely held to be of value as measures of different aspects of the scale or intricacy of a piece of source code. While McCabe’s cyclomatic number is well defined by its initial proposer, many different definitions exist of counting rules for the lines of code metric. One common variant (used in Boehm’s COCOMO estimation method among other places) is a definition that excludes blank and comment lines, but counts all others. Other variant rules include exclusion of non-executable lines (e.g. declaration statements). Another variant is source statements: for languages like FORTRAN and COBOL, the bulk of statements are expressed in single lines and there is a fairly close agreement between the number of statements and the number of lines.
With more modern languages like C, C++, Ada and Pascal, where multi-
statement lines become legal, multi-line statements become more common
and the agreement is less close. There is still, however, likely to be a
linear correspondence between the two counts across bodies of code which
are similar in style (e.g., written by the same programmer(s)).

Weyuker's original paper on properties of complexity measures considers
the McCabe metric and also statement count which, as her examples are
basically in FORTRAN, is to all intents and purposes a variant on the LOC
metric. She observes that both metrics satisfy properties 1, 3, 4, 5 and
8, both fail to satisfy properties 6, 7 and 9, and that property 2 is
satisfied by the statement count and not satisfied by cyclomatic
complexity.

The failure of a scalar reading of the McCabe's cyclomatic complexity
number to satisfy property 2 is due to the fact that the counting rules
for the McCabe metric are only sensitive to certain kinds of statements.
Arbitrarily large numbers of the statements that are not countable may be
added. Such statements include variable declarations and assignments.
Thus, given a single program that yields a particular value for the
cyclomatic complexity, it is possible to derive an infinite set of
differing programs which yield the same complexity, simply by repeatedly
inserting declarations and/or assignment statements.

A tool generating a response set consisting solely of measurements of
cyclomatic complexity over different parts of the submitted source code
would be subject to the same weaknesses, hence the response set notation
does not enable cyclomatic complexity in its own to satisfy property 2.
If we admit the use of a tool generating a response set containing observations of multiple metrics, we can introduce an alternate dimension to the view of the software generated by the tool, with another metric selected to illuminate the 'blind spot' caused by the insensitivities of MVG. For example, if we specify that the response set for our tool will consist of the LOC and MVG observations for each function definition processed, the arbitrary extensions of the program to which the MVG metric is insensitive will be clearly visible in the LOC metric. It is true that a response set consisting solely of LOC observations would also satisfy property 2. However, this does not imply that it would be as useful as the combined response set or even that it would be as useful as the (property 2 non-compliant) set of observations of MVG.

Property 6, which neither measure satisfies for scalar results, detects dependence of a metric on interaction between program components when combined. Weyuker observes that the reason for both of these measures failing to satisfy this property is that both are statically determined by the internal content of a component, and cannot be sensitive to the relationship between a particular component under measure and other components being measured at the same time. This constraint also applies to any response set based on LOC or MVG or any combination of the two, although either or both could be incorporated into a compliant response set with some other metric which is sensitive to inter-component relationships.

Property 7 considers the effectiveness of a metric in evaluating programs that differ from one another only in the order of statements. While both LOC and MVG fail to satisfy this property when considered as scalars, a
response set based on either measure can detect the rearrangement of a program by revealing the distribution of the LOC or MVG items between the measured scopes. This capability reflects the real usefulness of such information. If we have two versions of the same program which consists of 2000 LOC divided into 30 functions, the version where the distribution of LOC between functions was more uneven would probably be considered the more complex of the two (in terms of predicted maintenance difficulty for example).

Property 9 is also, like property 5, dependent on the interaction between measured components, and like property 5, neither LOC nor MVG nor any response set including only these two can satisfy it.

3.4.5.2 Structural metrics

The structural metrics selected for implementation were the three metrics in the information flow metric suite proposed by Henry and Kafura (1981, 1984) and further discussed and developed by Shepperd (1990). There are three metrics in the original set: fan-in, fan-out and the composite metric IF4, which is calculated from the first two. The fan-in and fan-out metrics are defined at the module level, and count the number of other modules which either send information to or receive information from the module being counted.

It is easy to show that properties 1, 3 and 8 are satisfied for each of these metrics, that is:

- not all programs yield the same result (property 1);
- there exist more programs yielding the same result which are different (property 3); and
• If two programs differ only in choice of identifiers they will yield the same result (property 8).

As with MVG, these metrics do not in themselves satisfy property 2. Once again, this is because there are statements in the language (again, local declarations and assignments) which can be added in arbitrary numbers without affecting the resulting count of FI or FO. As IF4 is a dependent function only of FI and FO, it is naturally subject to the same constraint. As before, any of these metrics could be applied to add value to a response set providing that other observations are compliant to this property.

In relation to property 4, the requirement is that there should exist two programs of exactly equivalent effect which yield differing values of each measure. When considering fan-in or fan-out as a scalar measure over a single subprogram as a whole, this property can not be complied with, as two subprograms which have equivalent external interfaces must have identical sets of fan-in and fan-out items. A response set of observations of fan-in and fan-out over individual subprograms or modules can be shown to comply: a trivial example of this would be the splitting of any subprogram into two parts in different modules, with data flowing between the two. Provided that the algorithmic content of the two parts is unchanged, we can see that the modified program is externally equivalent to the original, but the additional invocation causes information flow in either direction or both directions, hence giving a distinguishable response set.

Property 5 requires that the addition of source code to a project never causes a reduction in the measure yielded. It is relatively easy to
demonstrate this for the scalar case by considering the act of analyzing an increment of source code. There are no situations we might see in the source code which we can conceive of as reducing the number of non-local data flows, and therefore we are guaranteed that the measure may go up or remain constant, but never fall. When considering the application of this property over response sets, we need to specify the operation that will be used for magnitude comparison. Broadly, if the comparison operator is based on summation or on a policy of extremity, we are guaranteed that adding source code to our analysis will increase the number and the property holds, while if the comparison operation is based on some form of averaging, we have no such assurance.

Property 6 relates to the ability of a measure to capture the intensity of interaction between components. Property 5 asserts that composition of components must always increment the measured magnitude of a program by an amount greater than or equal to zero, the size of the increment being an indication of the degree of interaction between the components being brought together. Property 6 requires that the size of the interaction increment must depend on the content of the components, and not merely their individual magnitudes when measured separately.

Taking the example of FI, assuming comparison on the basis of summation:

If we have two programs P and R for which there are no use relationships from one to the other, we can be confident that

$$\text{FI}(P; R) = \text{FI}(P) + \text{FI}(R).$$

We then consider a third program Q, selected from the set of all programs for which

$$\text{FI}(Q) = \text{FI}(P).$$
The set of candidates for Q is intuitively infinite, and must contain some programs that mention modules defined in R. If we select one of these candidates which has a single mention (use relationship) for a module defined in R, we find that

$$FI(Q;R) = FI(Q) + FI(R) + 1$$

$$= FI(P) + FI(R) + 1$$

$$= FI(P;R) + 1.$$

This logic demonstrates that property 6 can be satisfied by a suitably chosen example for FI. Satisfaction for FO is trivial to show by analogy, as are the variants FIc, FIv, FOC and FOv. Given that IF4 and its variants are dependent on the respective FI and FO values, it is obvious that these measures also satisfy property 6.

Property 7 requires that textual transposition of source code should be capable of yielding different magnitudes. The structural metrics (as they have been implemented in this project) depend solely on the interfaces defined to modules; and, as a result, redistribution of source from the body of one module's method implementations to another would not cause change to the interface. It is difficult to conceive of textual transpositions that affect the interfaces of modules in a way that changes the values of the set of structural metrics without rendering the programs invalid.

Satisfaction of property 9 is demonstrable for all of the structural metrics by observing that the example programs Q and R and Q;R selected in our discussion of property 6 show the required behaviour (i.e., $|Q| + |R| < |Q;R|$). Indeed, it is evident that over a wide range of comparison operations, including all of those with reasonably intuitive
behaviour, satisfaction of properties 5 and 6 must inevitably lead to satisfaction of property 9.

3.4.5.3 Summary of analysis for procedural and structural metrics

The analysis above has attempted to demonstrate that the requirements posed by Weyuker's axiomatic approach to the validation of metrics are easier to meet when they are applied across a set of measurements. A set of measurements (described above as a response set) can report the results of more than one metric algorithm applied across different components of a body of code as a whole. Such a response set is able to satisfy these requirements where no single one of the individual metric algorithms does. Table 4 summarizes the findings of this section and demonstrates that all of the nine properties can be satisfied with a response set which incorporates the two groups of metrics considered.
Table 4: Which metrics satisfied which of Weyuker’s axioms

<table>
<thead>
<tr>
<th>Metric Tag</th>
<th>Properties satisfied</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>yes  yes  yes  yes  yes  no  no  yes  no</td>
</tr>
<tr>
<td>MUG</td>
<td>yes  no  yes  yes  yes  no  no  yes  no</td>
</tr>
<tr>
<td>FI/ED/IF4 and all variants</td>
<td>yes  no  yes  yes  yes  yes  yes  yes  yes</td>
</tr>
</tbody>
</table>
Implementation of a metric tool

4.1 Background

The practical part of the research project consisted of these activities:

• specification, design and implementation of a program to analyze source code and gather and report on a number of metrics;

• deployment of the developed analyzer in an industrial setting for a period of 3 months; and

• a survey of the users of the analyzer on subjects relating to metric use.

4.2 Chronology of the analyzer implementation

4.2.1 Initial specification

A skeleton specification for the analyzer was developed around the time of the preparation of the research proposal. This specification appears as Appendix A to this thesis. Key points covered by the specification included:

• The primary purpose of the analyzer is to provide a tool useful to programmers responsible for developing or maintaining the code analyzed.

• Dependencies on the platform, source language analyzed, output format and metrics calculated to be localized to allow substitution of alternative policies in each of these areas.

• Analysis is to be performed in response to a single command from the user.
- Elapsed time for analysis of a body of code is to be considerably less than the compile time for the same code (an informal target of 20% of compile time was laid down).

- The initial implementation is to parse the language C++ as defined by *The C++ programmer's handbook* (Lucas, 1992).

The initial specification laid down a short list of metrics to be implemented, including lines of code, lines of comment, McCabe's cyclomatic complexity, and the measures of coupling proposed by Rajaraman and Lyu (1992).

At an early stage in the practical part of the project, an industry partner was recruited to act as the primary location for testing of the analyzer in the field. The setting within which the analyzer was to be tested was a team developing a large real-time system in C++ using the Booch methodology (Booch, 1994), working with the Rational Rose C++ design/code generation tool. Between the start of the project and the end of the field trial, the size of this team varied in the range 5 to 20 people, mainly C++ programmers, but also incorporating a small number of specialists in other areas, including documentation, database administration and database user interface development. Experience of members of the team in industrial software development varied from a small number of months to fifteen years, but few members of the team had worked on a project using the Booch methodology before.

4.2.2 Prototype implementation

The process of design and implementation of the analyzer involved several iterations over a period of about three years, from July 1994 up to
September 1997. The specification referred to in section 4.2.1 had been laid down by the beginning of 1995. By the end of 1995 a fully functional prototype version of the analyzer had been implemented. The tool was called CCCC (short for 'C and C++ Code Counter'). This version, numbered 1.0, had the following attributes:

- It calculated lines of code, cyclomatic complexity and lines of comment.
- It reported these figures and ratios between the comment figure and the other two counts on a function-by-function basis with a summation over each source file.
- It defined thresholds for the expected ranges of the metrics calculated. These thresholds were used to implement emphasized output of abnormal values of any of the measures. The program supported colour output on ANSI-compliant terminal sessions and typeface bolding on Epson or LaserJet compatible printers.

The prototype version was released on the Internet by upload to the FTP site at sunsite.unc.edu, and was demonstrated to a number of members of the industry partner's team as a means of stimulating discussion within the team about metric-related issues.

4.2.3 Intermediate versions

Work on the analyzer continued through 1996 in parallel with the process of identifying metrics appropriate to the industry team (as described in Chapter 3). In February 1997 a new version, number 2.0.3 was released. This version differed from the earlier release in the following ways:
Output was done as a report in the Hypertext Markup Language (HTML), thus allowing issues like tabular layout and use of colour for emphasis to be handled by the third party browser used to read the report.

Metrics based on the Information Flow measures proposed by Henry and Kafura (1984) were implemented.

Analysis of a body of source code built up a database, which was used to generate a report after all source files have been read, allowing the report to present multiple views of the data with different sort and summation policies.

The report generated by version 2.0.3 consisted of 5 tables:

- a summary table showing totals for measures over all code processed in the current run;
- a procedural summary table showing the totals over each module of the lines of code, comment and cyclomatic complexity counts;
- a procedural detail table showing the total over each function of the same metrics;
- a structural summary table showing the fan-in, fan-out and information flow metric for each module; and
- a structural detail table showing the specific relationships which contribute to the fan-in and fan-out count.

This version was also released onto the Internet, and a WWW home page was created to publicize the project and distribute related materials. The tool was also uploaded to sunsite.unc.edu. The availability of the tool was publicized in a range of USENET news groups including comp.software.measurement and comp.software-eng.
In the succeeding months, a number of improvements were made to the analyzer as a result of reports on its performance (and non-performance) from users in the field who had downloaded the package from the web site. In addition to these improvements, work was done at this time on proving the source language independence of the analyzer by implementing alternative language recognition modules for the languages Java and Ada 95. On completion of the implementation of these modules, a decision was taken to define a baseline version of the analyzer to support the industrial deployment phase of the research. The baseline was numbered 2.1.1, and was released to the Internet on 15 August 1997.

4.2.4 The reference version

The baseline software was deployed at the industry site, although it was not heavily used in the initial weeks. One observation arising from those users who did use it was that when substantial bodies of software were analyzed (e.g. of the order of 50 pairs of header and implementation files), the HTML report generated caused the browser software used to consume excessive amounts of memory. A new release was prepared to fix this problem by generating the detailed reports on each module into a separate file, retaining a summary report on the project as a whole.

An additional set of metrics were also implemented in this release, based on the Chidamber and Kemerer work on a metrics suite for object-oriented design (Chidamber & Kemerer, 1994). This version also incorporated some changes to the names used for some metrics to bring the product into line with common usage in the literature.

The new version was numbered 2.1.2, and was publicized on the Internet on 6 October 1997. Further maintenance releases of the software have been
made since that time, and the latest version at the time of writing is numbered 2.1.4.

Appendix B presents the result of analyzing a small sample of source code using the version 2.1.4, and Appendix C reproduces the HTML user guide to the program which forms part of the distribution.

Appendix D presents an analysis of the download logs for the FTP archive associated with the projects. These logs show that considerable numbers of users downloaded various versions of the analyzer, including over 1000 downloads of version 2.1.4 in the three formats in which it was provided (precompiled for Linux or DOS, or source only).

4.3 Architecture of the analyzer

The baseline version of the analyzer consists of the following parts:

- a command line processing module which handles selection of files to be analyzed and options to control analyzer behaviour;
- a set of alternate front ends dedicated to language recognition and implementation of the counting policies of the tool (one front end per language, all conforming to a common interface to integrate with the rest of the tool);
- an internal database which holds all of the knowledge acquired by the front end(s) for a particular run; and
- an output module which reads the internal database and generates a set of HTML reports relating to the run as a whole, and to individual modules analyzed.
Of these components, the internal database, output module and command line processor were implemented in hand coded C++, while the front ends were coded using a public domain language recognition toolkit called PCCTS.

4.3.1 The command line processor

The command line processor is based on the simple pattern of a Unix-style character based tool accepting a string of arguments and options input from a keyboard through a shell program. The arguments are the names of source files to be processed, while the options enable configuration of the following aspects of analyzer behaviour:

- the location of a directory where configuration and support files are stored;
- the use of non-default language recognition front ends for particular groups of files (by default, the front end is chosen based on the filename extension for each file);
- generation of debug output including diagnostics on standard error and a dump of the internal database to files in the file system;
- selective output of a subset of the tables supported by the particular version; and
- acceptance of a file containing a list of source files to be analyzed as an alternative to specifying the names explicitly on the command line.

4.3.2 The language recognition front ends

The multiple front ends of the analyzer which recognize the languages C++, Java and Ada, are all based on parsers generated using a public domain set of tools called PCCTS. PCCTS, which is described by Parr and Quong (1995), consists of a pair of code generation tools plus a small
library of code to support the generated code. The two tools break the job of language recognition down into two phases. One tool, called ANTLR, generates a program to parse the language given a sequence of lexical tokens representing the terminal symbols of the language. The second tool, called DLG, generates a lexical analyzer, which breaks up an arbitrary stream of bytes into the tokens expected by the parser.

PCCTS provides similar facilities to the lex and yacc programs, which are in common use in the Unix programming worlds, and the grammar file format is broadly similar. As with lex and yacc, PCCTS provides the ability to embed actions in the language specification to be executed on the recognition of specified constructs either in the lexical analyzer or the parser. The main differences between PCCTS and the standard lex and yacc programs are summarized as follows:

- Yacc generates table driven parsers based on LALR analysis of the input grammar whereas PCCTS generates recursive descent parsers based on LL(k) analysis. LALR is an acronym which stands for Left Associative Left Reducing, while LL(k) is not a direct acronym, its significance is that parsing is performed Left to right, with decisions taken at the Left hand end of the text with k tokens of lookahead (Parr, 1993, pp. 25-29).
- Yacc and lex (as defined in the POSIX standard) generate either ANSI or Kernighan and Ritchie C, while PCCTS supplies switches on both the syntax analyzer ANTLR and the lexical analyzer DLG which provide for generation of C++.
- PCCTS supports automatic generation of abstract syntax trees.
• PCCTS supports specification of multiple token lookahead for the initial analysis and arbitrary (potentially infinite) lookahead for runtime resolution of ambiguities.

• While lex and yacc require separate input files, under PCCTS a single language specification is required which contains both the syntactic and lexical rules of the language.

While C++ output was a desirable feature, the main reason for choosing PCCTS for this project over the more standard lex/yacc combination was the LL(k) parser logic it implements. This family of parser logic supports construction of a language recognition facility that operates via recursive descent. Each syntactic category defined by a rule in the grammar file is recognized by a single recognition function in the generated parser. Each of these recognition functions has the same name as the syntactic category that it recognizes, and each consists of the following elements:

• calls to other recognition functions to recognize syntactic categories specified on the right hand side of the grammar rule which defines the current category;
• calls to functions to check and consume lexical tokens specified in the grammar rule; and
• actions embedded in the grammar.

As an example, if one had the grammar rule:

\[
\text{assignment_expression} : \text{lvalue ASSIGN}_{OP} \text{rvalue SEMICOLON} \quad \ll \text{nassigns++} \gg
\]
one might expect PCCTS to generate a function looking something like this:

```c
void assignment_expression(void)
{
    lvalue();
    ConsumeNextToken(ASSIGN_OP);
    rvalue();
    ConsumeNextToken(SEMICOLON);
    nassigns++;
}
```

This example (which does not attempt to reproduce the names of the functions or the error handling facilities generated by PCCTS) shows an easily understood structure for recognition of the assignment_expression category. By contrast, LALR parsing, as implemented in yacc, generates a table-driven state machine for recognition of the grammar specified. The code generated is nearly all table data, and there is no clear relation between the generated code and the grammar recognition context. The improved understandability of the code generated by PCCTS made designing appropriate actions to capture the required information from the source code easier.

Given the improved understandability of parsers built on the LL(k) parsing model, there is a cost in terms of language recognition power: the reason for the LALR parser being harder to understand is that it defers all interpretation of partially recognized structures until they become complete. This allows an LALR grammar to contain rules which are similar for any length of prefix, so long as no embedded actions are encountered before the point at which the two alternatives can be distinguished by examining a single token of the input stream. Conversely, where parsing is based on conventional LL(k) techniques, the rules of the grammar must be cast in such a form as to prevent the
occurrence of ambiguities between rules which cannot be distinguished using $k$ tokens of lookahead (analyses based on $k=1$ or 2 are reasonable, larger numbers lead to exponentially increasing analysis times for some grammars).

The use of multiple token lookahead in initial analysis and infinite lookahead where specified in the grammar are necessary to improve the performance of an LL($k$) grammar under conditions where the prefix of two rules are potentially ambiguous. These facilities are provided by PCCTS by the provision of a command line switch to control the number of tokens used for lookahead in the initial grammar analysis (the 'k' of the LL($k$)). PCCTS also defines the notation <<recognition_expression>>? called a syntactic predicate. This construct, which is applied as a qualifying clause for a particular alternative of an ambiguous rule, instructs the parser to scan forward in the token stream as far as is necessary to determine whether or not the syntactic category recognition_expression can be matched. If the match is successful, the associated alternative is selected, if not the parser rejects the alternative linked to the predicate and attempts to match the next viable alternative for the ambiguous rule. Syntactic predicates are used heavily in the C++ parser for CCCC.

4.3.3 The internal database

The data gathered was interpreted in the light of a simple relational data model that consisted of entities with the following names and definitions.
• Projects
A project is a collection of modules which are designed to work together as a software system to achieve some collective purpose.

• Modules
A module is a compound component of a software system which is characterized by a name and an associated set of data and methods which have high mutual cohesion and low coupling with data and methods outside the module.

• Members
A member is a single component of a module (usually either a member function or member attribute).

• Use Relationships
A use relationship is a feature of the implementation of a software system that implies some degree of inter-module coupling. A particular ordered pair of modules, \( \langle \text{supplier}, \text{client} \rangle \) uniquely identifies a use relationship record in the database although there may be more than one feature in the source code coupling the two modules (e.g. the same supplier and client may be related both by containment and parameter passing).

• Extents
An extent is the name coined for a textual section of source code in which the analyzer recognizes some attribute worthy of being stored in the database.

The neutral noun 'module' was chosen to represent a generic unit of language supported encapsulation. The initial implementation language was C++, and in conventional object-oriented C++ code we would expect the principal form in which modules appear to be C++ classes.
In other languages, it is possible that encapsulation would take other forms. Ada has packages, Java has packages and interfaces as well as classes, and C has none of these and relies on grouping functions by header or implementation files for any kind of modularization at a level between the individual function and the system as a whole. The data model is designed to support future versions of the analyzer which handle these and other languages.

The current model could be expanded to support Booch's concept of a class category relatively simply if the analyzer was configured to support some strategy for assignment of modules to class categories. In the case of C++ classes, such strategies could be developed based on the file system location of the source code files that define the class's interface or implementation.

Support for measurement of class categories within the analyzer would be a desirable enhancement because some writers support the proposal that the class category is a more desirable unit of encapsulation than the individual class. This is to say that when attempting to hide the implementation details of the classes which make up a particular class category, it is more important to hide details from classes in other class categories than it is to hide them from other classes within the same category.

Robert C. Martin discusses the issue of encapsulation at the class category level at some length, and presents a worked example of iterative refinement of an object-oriented design with intra-class category
coupling metrics treated as a significant driver of the refinement process (Martin, 1995).

4.3.4 The output module

As noted in section 4.3, the output of the analyzer is a set of reports in the hypertext markup language (HTML). These are viewed using a third party HTML browser after the analyzer run has completed.

The output module operates by iterating through the internal database calculating the various supported metrics at the project, module and method levels. Each scalar value calculated is associated with a character string tag that identifies what kind of measurement it is. For example, a measurement of the sum of McCabe's cyclomatic complexity over a module would be associated with the tag 'MVCm'. The analyzer accepts a configuration file that allows the user to define a 'treatment' to apply to measurements associated with each tag. The treatment controls the visible display of the calculated value including:

- the number of digits to be displayed after the decimal point;
- the background display colour triggered by different values of the measure; and
- a threshold for the numerator of ratio measures below which the measurement will be treated as not significant, and calculation and display of a value will be suppressed.

Of these three items, the control over the precision is an obvious cosmetic facility. The background display control is used to enable colour to be used to emphasize values which fall outside the user defined expected range. These entries in the output table identify the areas of
the project which are most subject to the risks associated with the particular metrics being displayed. The framework assumes that increasing value corresponds with increasing risk (this is reasonable for all of the selected metrics). The analyzer allows definition by the user of two thresholds to allow two levels of severity in the emphasis. Values that exceed the first threshold are highlighted in yellow, and are referred to in the analyzer source code as 'abnormal'. Values above the second value are highlighted in red, and are referred to in the source code as 'extreme'.

The ability to suppress display of ratio values depending on the numerator is useful to prevent the user's attention being distracted by spurious values based on small samples (e.g., the ratio of lines of code to comment within the body of a short function). It might be considered reasonable, for example, to establish a policy that on a function by function basis, there should be one line of comment for every five lines of code. This rule is easy to apply to long functions (say, over 10 lines), but becomes less meaningful with small functions. The suppression threshold allows the user to define a level below which the standard being specified does not apply, and where the ratio will not be calculated.

4.4 Data collection algorithms for metrics

In Chapter 3, where the basis for the metrics to be calculated by the analyzer were defined, each metric was specified by name and by reference to descriptions in the literature rather than laying down an exact description of the algorithms to be applied. The omission of rigorous algorithms at this stage was deliberate, as that chapter covered
theoretical considerations relating to the selection of metrics to be applied. It was anticipated that the exact algorithms to be applied would be constrained by practicalities as well as being driven by the theoretical considerations uncovered by the Goal/Question/Metric process.

Given the ability of the PCCTS generated front end to recognize constructs in any of the three supported input languages, actions were attached to the lexical and syntactic rules of the grammar to support gathering of data from which the required metrics could be derived. Broadly, the responsibilities of the components in relation to the data gathering are as follows:

The parser is central to the data collection strategy. It is responsible for recognizing stretches of code (corresponding to the 'extent' entity described in the section describing the analyzer’s internal database in section 4.3.3), and for recording data about these extents to the internal database. Three different functions are used to add extents to the internal database, one each for modules, methods and use relationships. Each extent record contains:

- identification of the module, method or use relationship the extent relates to;
- filename and start line of the extent (finish line would be available but is not stored at present);
- a set of flags giving information about issues like the prevailing visibility at the point in the code where the extent was seen; and
- a buffer of measures associated with the extent which were calculated in the lexer.
The lexical analyzer is also important in the collection of data: the procedural measures collected by the analyzer are all counted by identifying lexical patterns. The exact patterns counted for each of these metrics are described in the section on calculation of procedural metrics in section 4.4.1. The structural and object-oriented metrics are (as has been noted before), rather higher-level than the procedural metrics, and these are calculated across the generated database as a whole rather than being counted at the level of an individual extent.

4.4.1 Calculation methods for procedural measures

The LOC measure is counted by having a flag in the lexer that records whether or not a 'real' (non-comment, non-whitespace, non-preprocessor) token has been seen since the start of a line. When the end of line is recognized, if this flag is true, the LOC count is incremented, otherwise it remains the same. Either way, before processing the next line the flag is reset to false.

The COM measure is incremented every time the end of a C++ style comment ('//') is seen, at the end of every line and the closing delimiter within a C style comment ('/* ... */'). The lexer was configured to distinguish between 'normal' C++ style comments, and comments starting with the string '//'##', as the latter pattern is used by the Rational Rose design/code generation tool to embed information in generated source files. As this information is not true programmer supplied commenting, it is not credited to the COM count.

The MVG measure is incremented every time a token which implies a decision point is seen. In C++ the tokens in question are if, while,
for, until, &,&, ||, switch, and break. The use of token counts to approximate MVG is a widespread practice, for example see the example program presented by Conte, Dunsmore and Shen (1986, pp. 40-41).

Although the practice of calculating MVG by counting decision point tokens is common, it does not yield accurate results for all legal code. In most cases the differences between outcomes of token counting and hand calculation of the cyclomatic complexity are likely to be small providing that the measured code conforms reasonably well to the tenets of structured programming.

Logical operators are included in the list of tokens associated with MVG because they are defined as performing short cut evaluation, i.e. the second operand will not be evaluated if the value of the first operand is sufficient to deduce the outcome of the operation.

For switch statements, the MVG count will be correct if and only if the code block for each alternate outcome ends in a break statement. If multiple labels are used for the same block there is no problem, but allowing some labels to 'fall through' from their own (non-empty) block of code into the block associated with other labels will lead to miscounting. The switch keyword is counted as a proxy for the default case, which is held to represent a possible code path, whether or not it has an associated label.

The fact that the three procedural measures are recognized in the lexical analyzer rather than the parser causes a certain amount of inconvenience. They are communicated to the parser by letting every token which passes
from the lexer to the parser carry copies of running counts of each of the three quantities since the start of the current file. When the parser identifies an extent, it calculates the amount of each of these measures to allocate to the current extent by taking the difference for each count between the first and last token included in the extent. The count for LOC is incremented by one to account for the closure of the line on which the final token of the extent is seen.

This method of calculation leads to a small level of miscounting due to overlapping extents. For example a class definition containing declarations of a number of member functions will give rise to a module extent record covering the whole of the class definition, and method extent records for each member function declared. When the counts associated with these extents are summed to obtain the total LOC associated with the module the lines within the member function definitions will be counted twice. If header files alone were processed, this might give rise to significant inaccuracies, but for a typical C++ project where implementation files are processed as well as headers the error due to this cause would be unlikely to be more than 5%.

The likelihood of the various inaccuracies described above was mentioned in the analyzer documentation. Feedback received through email and responses to the free text question in the project survey (the latter reproduced as Appendix G) indicated that inaccuracies at the level of 3-5% were widely regarded as acceptable. Some users reported higher levels of inaccuracies. On investigation, it was found that these cases were usually due to the failure of the analyzer to parse the idioms used in the submitted code. In most cases this was either because the code was
of the unsupported Kernighan and Ritchie C dialect, or because the preprocessor was required to make the parts of the code analyzed by CCCC conform to the syntax rules of C++.

The decision not to require the preprocessor to run on code as part of the analysis process was prompted by a number of factors:

- The preprocessor flags would need to specify a full list of directories where C/C++ header files for the project are to be found, making the tool cumbersome to run by hand, especially if it was desired to analyze multiple files which require differing inclusion path specifications.

- The purpose of the tool is to analyze human readable source code. The preprocessor is the first of a series of processes that run to convert the human readable form of the program into a machine executable one. While the preprocessor output file is still human readable, it is different from the primary artifact we wish to analyze.

4.4.2 Calculation methods for structural measures

The structural measures are calculated by traversing the internal database’s table of relationships for each module, identifying all relationships the module participates in as a supplier or a client. Each relationship record has a number of subordinate extent records that represent the low-level code features that show that the relationship exists. In the reference version of the analyzer the extents recorded are inheritance lists, data member declarations and method declarations and definitions.
Each individual extent is qualified with a vector of flags, which represent (among other things), the visibility of the extent, and whether the supplier in the relationship is specified by reference or by value. The values of these two flags across the set of extents associated with a single relationship are combined. The combined visibility flag is calculated on the basis that the visibility of the relationship is equal to the visibility of the most visible extent that contributes to the relationship. The combined flag for specification by reference is set if all of the extents have this flag set.

Once the extent flags for each of a module’s relationships have been combined in this way, the structural metrics can be calculated. The fan-in (FI) is the number of relationships in which the module operates as a client, while the fan-out (FO) is the number of relationships in which the module operates as a supplier. These two numbers are combined into the information flow metric IF4 using the formula:

$$IF4 = (FI \ast FO)^2$$

The visibility and specification by reference flags are used to calculate two variant sets of these measures. For the visibility variant, only relationships for which the visibility is public, protected or not known are counted, and the modified measures are denoted FIV, FOV, and IF4V. Unknown visibility occurs where the existence of a method is known from seeing its implementation, but its visibility is not known because its declaration as part of the class interface has not been seen. The by-reference flag is taken to indicate the fact that the coupling created by the relationship can be satisfied by a forward declaration of the supplier class. If this flag is set for the relationship as a whole, the
relationship is excluded from the second modified count, which gives rise to the measures denoted FLt, NOc, and IF4c.

4.4.3 Calculation methods for Chidamber and Kemerer's measures

The measures defined by Chidamber and Kemerer are fairly simple to calculate, given that they are quite closely related to the features expected of an expression of the object-oriented paradigm in the main target programming language. The only anticipated inaccuracies relate to situations where the analyzer operates on incomplete data. For example, consider the case where if a particular run has processed the definition of a class which inherits from the system library class ifstream, but the definition of ifstream has not been processed. In this case the measurement of DIT (Depth of Inheritance Tree) will not include layers of inheritance above ifstream in the hierarchy.

4.5 Potential enhancements identified

At the time of writing, the most recent released version of the analyzer is 2.1.4, which differs from the reference version described above only by a few bug fixes. In the light of experience with the analyzer and suggestions from users in the field, a number of areas where enhancements could be made have been identified. Some of the more significant ones are as follow:

- The double counting of lexical measurements identified in section 4.4.1 could be eliminated or ameliorated by an improvement to the lexer/parser counting communication mechanism.
- The analyzer's data model and data gathering facilities could be expanded to identify class categories and allocate modules to them. This would be particularly useful for improving the meaningfulness of
the structural metrics extracted from Ada source code, as the Ada package construct provides a syntactic expression of a class category, which is completely ignored by the present version of the analyzer.

- The structural metrics described in Robert C. Martin's work described in section 4.3.3 were not considered for use in the implementation part of the project because they were not known to the experimenter at the relevant time.

- The analyzer could be enhanced to support saving the internal database in some format. This would enable incremental analysis of large bodies of code, and also might enable the creation of alternative user interfaces for accessing and exploiting the data gathered.

- Since the implementation of version 2 of the analyzer, the C++ language has undergone the final stages of a process of standardization, and additional syntax features and a standard library for the language have been defined. The parser for C++ needs to be expanded to handle the newer syntax and keywords.

- The standardization of the language and its associated libraries has also created an opportunity for improving the portability of the analyzer by using the classes now defined in the standard library for character strings and the standard template library to improve database performance.
5 **Practitioner opinion survey**

5.1 **Background**

This section of the thesis will deal with the design and administration of the survey of software practitioners performed as a part of the research project.

5.2 **Goals of the survey**

The goals of the survey are to contribute to our knowledge about the answers to the research questions listed in Chapter 1. Specifically, the survey is intended to investigate:

- practitioners' opinions on the value of metric tools in general;
- the value of specific metrics implemented in this study;
- dangers associated with the use of metrics;
- helpful and unhelpful ways of presenting metrics; and
- the value of specific metrics as predictors of specific risks.

The survey also seeks to isolate as independent variables aspects of the context in which the tools are deployed including:

- the employment status, industry sector and number of years of experience of each respondent;
- the platforms, application areas, languages, design methodologies and techniques with which each respondent is familiar;
- the personal culture of the individual practitioner;
- the culture of the organization within which the practitioner works; and
- the immediate goals of the practitioner in using a metric tool.
The motivation for isolating these aspects is that the practical value of metric techniques as a whole, and of different specific metrics, may well be sensitive to these variables. Identification of such links would help individuals and organizations predict the benefit or otherwise of specific approaches to the metric issue.

5.3 Factors under investigation

5.3.1 Input factors

One of the goals of the study is to link the validation of the use of metric tools in general to the context in which the tool is used. The survey questionnaire will attempt to categorize respondents according to the context of use at three levels:

- **Personal context**
  
The survey asks questions about the primary source of income of the respondent, the sector of the industry in which he/she is employed, the number of years of experience, and about his/her exposure to some of the major application areas, platforms, languages, methodologies and techniques. There are also some questions which attempt to classify the cultural background of the respondent on the Greek/Roman continuum identified in section 2.7.

- **Organizational context**
  
The survey contains a number of questions that attempt to classify the organization within which the respondent works on the Greek/Roman continuum.

- **Application context**
  
The survey asks a number of questions that are aimed at identifying the particular goals for which the respondent uses metric analysis tools. The questions are geared to identifying the distinct possible
applications of metric tools (e.g. for the respondent to check his/her own work, to check the work of a subordinate, etc.). This section of the questionnaire also raises the issue of the degree of trust that automated analysis of this kind will not be used against the respondent's interests within his/her organization.

5.3.2 Outcome factors

For each of the three groups of metrics selected for implementation there is a set of questions geared at discovering whether the respondent finds the metrics in the group useful.

There is also a group of questions which investigates the usefulness or otherwise of some specific features of the implemented metric tool, including aspects of the tool's presentation of results and the modified metrics IF4c, IF4v and WMCv.

5.4 Survey Design

The design of the survey is constrained by the following goals:

- the survey instrument to be as straightforward as possible;
- outcomes to be suited to automatic processing; and
- completion by an individual respondent to be performed within a 15 minute period.

These goals are intended to support the attraction of as wide a body of responses as possible.

The questionnaire was divided into a number of sections with the following topics:
• factual data about the respondent including years of experience, employment sector, size of employer, what kind of projects the respondent works on, the respondent's responsibilities, and the languages, methodologies and paradigms in use at the respondent's workplace;
• questions relating to the respondent's personal culture;
• questions relating to the culture of the respondent's employer organization;
• questions relating to the ways the respondent uses metric techniques;
• questions relating to the usefulness or otherwise of specific metrics for characterizing particular attributes of source code;
• questions relating to the usefulness or otherwise of some of the presentation strategies adopted by CCCC; and
• a free format field for entry of comments on the issue which the respondent might have.

Aside from the email address and free format comment fields, all of the questions were posed in one of three formats:

• Three of the factual questions offer the respondent a selection of a number of question specific responses from which he/she must select one.
• The remainder of the factual questions concern the presence or absence of a particular factor (e.g. a particular responsibility, or use of a particular language). These questions allow only a yes/no answer.
• The opinion-based questions all had the same form of a statement to which the respondent must indicate agreement or disagreement on a 5 point scale. The points on the scale were identified on the questionnaire form with the following words:
• disagree strongly;
• disagree;
• neutral;
• agree; and
• agree strongly.

The five response groups to the opinion questions were represented in the stored form of the questionnaire responses by the numbers 1 to 5, 1 being 'disagree strongly'. The choice of a five point scale was motivated by the following factors:

• The use of an odd number of options allows the provision of a neutral option as the midpoint of the scale, whereas the alternatives (either not providing a neutral option, or providing a different number of options above and below the neutral point) have the potential to bias the study.

• A three point scale might have been too coarse as to lose meaningful data.

• While more than five points on the scale might have allowed collection of data with higher differentiation between respondents, it was regarded as unlikely that the additional data obtained would add to the significance of the overall results.

Discussion of the framing of individual questions will be handled in the section dealing with those questions in chapter 6.

5.5 Survey administration and processing

The questionnaire was implemented as a JavaScript HTML page, and respondents were able to complete it over the Internet, with a CGI script causing a response file to be written to a directory on the University's web server for each respondent who completed the survey.
With the exceptions of a field in the factual section for a return email address, and the free format comment field, all questions are implemented as HTML forms controls which allow the user to select between a fixed range of options. In all cases the initial state of the control was set to correspond to a neutral or agnostic reply.

The survey instrument was reviewed by the researcher's academic supervisor, and also by key members of the industry partner's team and other researchers in the area with whom the researcher was corresponding by email.

Data collection based on the final version of the survey instrument began in January 1998 and continued until the final data was harvested in September of the same year. A decision was taken to eliminate any responses for which all of the opinion responses were neutral, on the grounds that these probably represented casual web surfing rather than considered responses. A total of 24 valid responses was received.
6 Survey outcomes

6.1 Background

This chapter discusses the framing of the questions asked in each section of the survey, tabulates the responses received, and discusses the findings. As far as possible the tables show the exact wording of each question on the form, and all options presented on the form are listed even if no respondents selected them.

The survey instrument consisted mainly of questions with a finite number of predefined answers. These fell into three groups:

- three general questions aimed at assessing the demographics of the respondents as a group (employment status, employment sector, years of experience);

- forty-eight binary (yes/no) questions aimed at identifying the work responsibilities, development platforms, languages, methodologies and techniques of each respondent; and

- forty-five questions aimed at identifying each respondent's agreement or disagreement with a set of statements about their working environment, software metrics and the relationship between the two.

In addition to these closed questions, there was a single free-format field provided on the survey form for respondents to add any observations they wished to make, and another field to allow them to supply their email address.
The survey instrument was created as an HTML form and posted on the University's departmental web server as a link under the home page describing the project, and was publicized at various times in the relevant USENET news groups (primarily comp.software.measurement, also comp.software-eng). It was planned to allow the survey to run for a minimum of three months, while the earlier chapters of this thesis were being written. In the event, responses were collected over a period of approximately 9 months from January until September 1998. Over this period of time a total of 24 valid responses were received. A representation of the HTML survey form appears as Appendix E, and a tabulation of the raw responses appears as Appendix F.

Processing of the responses was automated to minimize the danger of calculation error. The responses were collected from the directory into which the web server wrote them, and a script in the language Perl was run to convert the responses into a set of text files in a comma separated format suitable for import into the MS Access 97 database package. Within Access, a number of views and reports were set up to summarize the data in convenient format. The tables in the remainder of this chapter have been generated by Access, before being imported into the current MS Word 97 document.

The email reply addresses and free-format comments were not processed in this way. The free-format comments were examined to determine whether any appeared to contain material that might be sensitive or confidential. A small number of minor comments which might help to identify the respondents were removed, allowing all of the comments submitted to appear in Appendix G of this thesis.
6.2 General questions

The research project is primarily concerned with evaluating conditions in the industrial software engineering environment, hence it was important to verify that the respondents are in fact basing their opinions on experience in such a setting. The first three questions on the survey instrument were aimed at verifying that the demographics of the respondents were representative of people working in the target environment.

The three questions in this section focused respectively on the primary source of income of the respondent, their employment status and number of years experience in the software industry.

6.2.1 Main source of income

We expect a degree of exchange of personnel between industry and the academic world, and also a proportion of practitioners from industry who are engaged in some form of study or teaching. The wording of the question is engineered to separate respondents who are solely involved in the education sector from those who have a divided affiliation. The question was biased in this way as there was an expectation that a proportion of the respondents working in industry would also be involved in some form of academic life. Such respondents are clearly part of the target group which this project seeks to investigate, in a way that respondents who are solely students or teachers are not.

Table 5: Source of income

<table>
<thead>
<tr>
<th>Main source of income</th>
<th>Education</th>
<th>Government</th>
<th>Private Sector</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of respondents</td>
<td>3</td>
<td>2</td>
<td>18</td>
<td>1</td>
</tr>
</tbody>
</table>
The responses to this question confirm that the respondents are overwhelmingly in the target group of practical software developers.

6.2.2 Employment status

This question takes an alternate view of the affiliation of the respondent, asking him/her to identify with one of a set of labels relating to employment status.

Table 6: Employment status

<table>
<thead>
<tr>
<th>Employment status</th>
<th>Contractor</th>
<th>Educator</th>
<th>Employee</th>
<th>Researcher</th>
<th>Student</th>
<th>Un-employed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of respondents</td>
<td>3</td>
<td>2</td>
<td>15</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The responses to this question are consistent with the previous question, the overwhelming majority of respondents identify themselves as either contractors or employees, rather than in any of the education-related groups (or unemployed).

6.2.3 Years of experience

This question asks the respondents to report their number of years experience in software development.

Table 7: Length of experience

<table>
<thead>
<tr>
<th>Years of employment in software development</th>
<th>0 to 1.99</th>
<th>2 to 4.99</th>
<th>5 to 9.99</th>
<th>10 to 14.99</th>
<th>15 or more</th>
<th>not given</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of respondents</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>5</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

The responses to this question indicate that the respondents are concentrated in the upper reaches of the experience scale with a median experience of over 10 years.
6.2.4 Summary for general questions

Overall, the indications from these three questions are that the respondents to the survey are indeed representative of the population whose attitudes and opinions this project sets out to investigate. They are overwhelmingly concentrated in practical industry positions, and have significant periods of industry experience to inform their judgements.

Consideration was given to excluding some of the respondents from further processing on the basis of their replies in this section. Specifically, these were the two educator respondents, the one student, the one unemployed and the two respondents with less than two years experience. Removing these respondents would have reduced the number of responses from 24 to 18, it was decided that in view of the low response rate not to exclude any of the respondents.

6.3 Binary questions

The next set of questions consist of three groups of yes/no response questions which ask each respondent about his/her job responsibilities, areas of practice in terms of application domain and operating platforms and languages, methodologies and techniques in use. These questions are intended to categorize further the respondents as a group. They may also be used for investigation of second order effects (e.g. to answer questions like "Do respondents using an object-oriented methodology have significantly different opinions on the value of the object-oriented metrics?").
6.3.1 Responsibilities

The question on responsibilities aims to cover as wide a range as possible of the individual activities that make up the software development process.

Table 8: Responsibilities

<table>
<thead>
<tr>
<th>Activity</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis</td>
<td>62.5</td>
</tr>
<tr>
<td>Specification</td>
<td>50.0</td>
</tr>
<tr>
<td>Design</td>
<td>70.8</td>
</tr>
<tr>
<td>Programming</td>
<td>75.0</td>
</tr>
<tr>
<td>User documentation</td>
<td>20.8</td>
</tr>
<tr>
<td>Technical documentation</td>
<td>33.3</td>
</tr>
<tr>
<td>Maintenance</td>
<td>45.8</td>
</tr>
<tr>
<td>Testing</td>
<td>58.3</td>
</tr>
<tr>
<td>Design Reviews</td>
<td>50.0</td>
</tr>
<tr>
<td>Code Reviews</td>
<td>75.0</td>
</tr>
<tr>
<td>Document Reviews</td>
<td>25.0</td>
</tr>
<tr>
<td>Quality Assurance</td>
<td>41.7</td>
</tr>
<tr>
<td>Estimate or Tender Preparation</td>
<td>25.0</td>
</tr>
<tr>
<td>Project Management (Commercial)</td>
<td>16.7</td>
</tr>
<tr>
<td>Project Supervision (Technical)</td>
<td>41.7</td>
</tr>
<tr>
<td>Staff Appraisal</td>
<td>20.8</td>
</tr>
</tbody>
</table>

The responses to this question tend to suggest that the majority of the respondents are more involved in practical day-to-day software development than in supervisory, management or marketing roles. Note that the survey instrument did not contain detailed definitions of any of these terms, different respondents may have different activities in mind in some cases (notably Quality Assurance).

6.3.2 Languages, methodologies and techniques

The next group of questions asked about the languages, methodologies and other theoretical techniques recently used by the respondent.
Amongst the language responses, C++ is the most popular. The next most popular languages are C and Java, both closely related to C++. After these languages come Ada and Pascal/Delphi, both in the 20-25% range, followed by Eiffel, Cobol and Fortran, each with 4.2% (i.e. a single respondent).

The dominance of C++ is no great surprise, given that it is the primary language of the CCCC tool, and is regarded as the most widely used language in the software industry at present. The frequency of C and Java are also explainable in terms of the relationship of each to C++, C as the progenitor of C++ and its predecessor as most widely used language, and Java as C++'s offspring and a contender to be the 'next big thing'. On average each developer declared involvement with just over 2.5 languages.
The questions on methodology revealed that the clear, but not dominant, leader in the area is the Boodh methodology. A brief examination of the underlying raw data revealed that the 21 positive responses to the different methodology questions come from 21 different respondents, that no respondent reported using more than one methodology, and that only 3 respondents did not claim to have used any object-oriented methodology at all.

This group of questions also included questions on two additional techniques, the use of formal methods and the use of design patterns. Significant minorities of the respondents had used both techniques. The level of 25% of respondents who claimed to have used design patterns is not particularly striking in view of the current interest in the field. The 30% figure for experience with formal methods was more surprising, given that application of these techniques is not believed to be widespread outside a few specialized application domains. It is possible that this surprising result is due to misinterpretation of the phrase "formal methods" (which was intended to cover rigorous program proving techniques) as meaning "formally defined inspection processes".

6.3.3 Application areas and platforms

The next group of questions asked which application domain area(s) the respondent had recently worked in and also what operating system platform(s) they had been using over the same period of time.
The picture that emerges from this question is that while no single application domain dominates, the proportion of financial/transaction processing applications is fairly low. The relatively high number of developers involved in development tools is interesting; perhaps this reflects a professional interest in the CCCC software as a tool for developers. On the operating system side, there are no major surprises. Unix and DOS/Windows are found to be the dominant platforms. Given that there are more DOS/Windows desktops than Unix across the market, one might have expected DOS/Windows to have outpolled Unix. There are at least two possible explanations for the current finding:

- much DOS/Windows software is from the shrink-wrapped market, where there is likely to be a lower ratio of developers per desktop; and
- the respondents are likely to be people who have experimented with the CCCC program, and while a Win32 version was released, it was basically

<table>
<thead>
<tr>
<th>Which of the following application areas and platforms have you worked in over the last two years</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Financial Applications</td>
<td>20.8</td>
</tr>
<tr>
<td>Online Transaction Processing</td>
<td>8.3</td>
</tr>
<tr>
<td>Other Database Applications</td>
<td>41.7</td>
</tr>
<tr>
<td>Realtime/Control/Embedded</td>
<td>50.0</td>
</tr>
<tr>
<td>Operating Systems</td>
<td>20.8</td>
</tr>
<tr>
<td>Development Tools</td>
<td>41.7</td>
</tr>
<tr>
<td>Productivity Tools</td>
<td>20.8</td>
</tr>
<tr>
<td>Client/Server Applications</td>
<td>41.7</td>
</tr>
<tr>
<td>DOS/Windows</td>
<td>67.5</td>
</tr>
<tr>
<td>OS/2</td>
<td>14.5</td>
</tr>
<tr>
<td>Macintosh</td>
<td>4.2</td>
</tr>
<tr>
<td>Unix and similar</td>
<td>70.8</td>
</tr>
<tr>
<td>VMS</td>
<td>4.2</td>
</tr>
<tr>
<td>IBM Mainframe</td>
<td>0.0</td>
</tr>
<tr>
<td>Other Mainframe</td>
<td>0.0</td>
</tr>
<tr>
<td>Other Microcomputer</td>
<td>8.3</td>
</tr>
</tbody>
</table>
a port of the Unix version, and was widely found to be less satisfactory than the original.

It is interesting to look at the totals of percentages across each of the categories; these indicate that each respondent is involved on average in around 2.4 of the specified application domains, and 1.6 of the specified platforms.

6.3.4 Summary for binary questions

While the general questions were useful in confirming that the respondents as a whole are active in the software engineering industry, the binary questions are intended to characterize the respondents within the industry.

One of the goals of the project as a whole was to collect data on the interaction of metric tool use and workplace culture. The Greek/Roman dichotomy as discussed in the literature by Robert L. Glass and others which was introduced in section 2.7 is one of the frameworks against which we hope to be able to make observations, and this group of questions starts to provide some data in this area.

In the questions on responsibilities, most of the activities which might be classified as belonging to the Roman end of the spectrum (user and technical documentation, document reviews, estimate and tender preparation, commercial project management, staff appraisal) were all minority interests. The one exception to this pattern was the quality assurance item, which, while still a minority at just over 40%, would seem to be a significant area of interest for the respondents.
In the application domain questions, one might classify the financial and online transaction processing domains as predominantly Roman, and the real-time/control embedded area as Greek (the others are more difficult to classify). Again in these cases, the pattern that emerges is that the areas identified as belonging to Romans are underrepresented.

6.4 Agreement questions

As noted in section 5.4, the rest of the questions (with the exception of the final free-format comment field) were all cast in the format of a statement to which the respondent was asked to indicate a level of agreement or disagreement on a five-point scale. While the points on the scale were labeled verbally on the survey instrument, they are represented numerically in the electronically captured survey response data.

The numeric representations of the five response options have been retained through the processing. This implies an assumption that the verbal categories presented to the respondents can validly be transformed to a ratio scale. The language used for the category descriptions and the format of presentation of the options in the questionnaire (i.e. in their mapped numeric order) make this a reasonable assumption in the current case. Accepting the validity of the numerical mapping, the figures have been used to calculate the mean, variance and standard deviation of the distribution of responses among the five categories for each question.

6.4.1 Preferred ways of working

The first of the groups of agreement questions covers issues of personal preference in relation to working environment.
Table 11: Preferred ways of working

<table>
<thead>
<tr>
<th>Opinions on the respondent's preferred ways of working</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>MEAN</th>
<th>SD</th>
<th>VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1 I like to choose my own tools</td>
<td>0.0</td>
<td>8.3</td>
<td>8.3</td>
<td>58.3</td>
<td>25.0</td>
<td>4.00</td>
<td>0.83</td>
<td>0.70</td>
</tr>
<tr>
<td>A2 I prefer working in a team to working alone</td>
<td>0.0</td>
<td>12.5</td>
<td>45.8</td>
<td>29.2</td>
<td>12.5</td>
<td>3.42</td>
<td>0.88</td>
<td>0.78</td>
</tr>
<tr>
<td>A3 I would like to spend more time working on</td>
<td>4.2</td>
<td>37.5</td>
<td>33.3</td>
<td>25.0</td>
<td>0.0</td>
<td>2.79</td>
<td>0.88</td>
<td>0.78</td>
</tr>
<tr>
<td>documentation</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A4 I don't mind depending on other people</td>
<td>0.0</td>
<td>12.5</td>
<td>41.7</td>
<td>37.5</td>
<td>8.3</td>
<td>3.42</td>
<td>0.83</td>
<td>0.69</td>
</tr>
<tr>
<td>A5 I like learning new skills better than exercising old ones</td>
<td>0.0</td>
<td>4.2</td>
<td>29.2</td>
<td>50.0</td>
<td>16.7</td>
<td>3.79</td>
<td>0.78</td>
<td>0.61</td>
</tr>
</tbody>
</table>

The results in Table 11 contain no major surprises. Results for questions A1 and A3 tend to confirm the previous observations about the cultural background of the respondent population being more likely to be near the 'Greek' end of the Greek/Roman scale. Results for questions A2 and A4 reveal that, while there is a marginal positive trend for the two questions relating to attitudes to team working, for both questions there is a majority which is either neutral or marginally opposed to the proposition. There is also a marked preference for work involving new skills. It would be interesting to know whether this preference is representative of the wider population of software engineers of comparable experience.

6.4.2 Professional and social environment

The next group of questions are focussed on similar issues to the first group, except that they are designed to identify attributes of the environment in which the respondent actually works (as opposed to his/her preferences for those attributes).
Table 12: Professional and social environment

<table>
<thead>
<tr>
<th>Opinions on the respondent's professional and social environment</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>MEAN</th>
<th>SD</th>
<th>VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1 My workplace encourages developers to choose their own tools</td>
<td>4.2</td>
<td>16.7</td>
<td>29.2</td>
<td>33.3</td>
<td>16.7</td>
<td>3.42</td>
<td>1.10</td>
<td>1.21</td>
</tr>
<tr>
<td>B2 My workplace values talent higher than conformity</td>
<td>0.0</td>
<td>16.7</td>
<td>16.7</td>
<td>50.0</td>
<td>16.7</td>
<td>3.67</td>
<td>0.96</td>
<td>0.93</td>
</tr>
<tr>
<td>B3 My workplace gives me sufficient freedom</td>
<td>0.0</td>
<td>16.7</td>
<td>16.7</td>
<td>41.7</td>
<td>25.0</td>
<td>3.75</td>
<td>1.03</td>
<td>1.07</td>
</tr>
<tr>
<td>B4 My workplace gives some of my colleagues too much freedom</td>
<td>8.3</td>
<td>29.2</td>
<td>54.2</td>
<td>4.2</td>
<td>4.2</td>
<td>2.67</td>
<td>0.87</td>
<td>0.75</td>
</tr>
<tr>
<td>B5 My workplace applies (technical) review techniques to good effect</td>
<td>16.7</td>
<td>20.8</td>
<td>29.2</td>
<td>29.2</td>
<td>4.2</td>
<td>2.83</td>
<td>1.17</td>
<td>1.36</td>
</tr>
</tbody>
</table>

Again, the results for questions B1, B2, B3 and B4 all support the proposition that the respondents are drawn from the Greek end of the Roman/Greek scale. The negative response to question B5 is the only one which could be seen as reflecting negatively on the respondents' feelings about their workplace.

6.4.3 General metrics issues

These questions cover a range of overall attitudes towards metrics and their use.
Table 13: General metrics issues

<table>
<thead>
<tr>
<th>Opinions on metrics and metrics issues (General Issues)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>MEAN</th>
<th>SD</th>
<th>VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1 I use software code metrics to evaluate software design</td>
<td>12.5</td>
<td>20.8</td>
<td>33.3</td>
<td>33.3</td>
<td>0.0</td>
<td>2.88</td>
<td>1.03</td>
<td>1.07</td>
</tr>
<tr>
<td>C2 I use software code metrics to evaluate implementations</td>
<td>4.2</td>
<td>8.3</td>
<td>29.2</td>
<td>30.0</td>
<td>8.3</td>
<td>3.50</td>
<td>0.93</td>
<td>0.87</td>
</tr>
<tr>
<td>C3 I use software code metrics on my own code</td>
<td>0.0</td>
<td>8.3</td>
<td>20.8</td>
<td>54.2</td>
<td>16.7</td>
<td>3.79</td>
<td>0.83</td>
<td>0.69</td>
</tr>
<tr>
<td>C4 I use software code metrics on code written by my current peers</td>
<td>4.2</td>
<td>16.7</td>
<td>37.5</td>
<td>29.2</td>
<td>12.5</td>
<td>3.29</td>
<td>1.04</td>
<td>1.09</td>
</tr>
<tr>
<td>C5 I use software code metrics on code written by my current subordinates</td>
<td>8.3</td>
<td>16.7</td>
<td>45.8</td>
<td>25.0</td>
<td>4.2</td>
<td>3.00</td>
<td>0.98</td>
<td>0.66</td>
</tr>
<tr>
<td>C6 I use software code metrics on code written by someone else which I now maintain</td>
<td>8.3</td>
<td>12.5</td>
<td>41.7</td>
<td>20.8</td>
<td>16.7</td>
<td>3.25</td>
<td>1.15</td>
<td>1.33</td>
</tr>
<tr>
<td>C7 I use software code metrics to identify areas for review</td>
<td>0.0</td>
<td>8.3</td>
<td>33.3</td>
<td>45.8</td>
<td>12.5</td>
<td>3.63</td>
<td>0.82</td>
<td>0.68</td>
</tr>
<tr>
<td>C8 I use software code metrics as an input to personnel appraisal processes</td>
<td>16.7</td>
<td>29.2</td>
<td>37.5</td>
<td>12.5</td>
<td>4.2</td>
<td>2.58</td>
<td>1.06</td>
<td>1.12</td>
</tr>
<tr>
<td>C9 I use software code metrics to help estimate effort on future projects</td>
<td>0.0</td>
<td>25.0</td>
<td>25.0</td>
<td>29.2</td>
<td>20.8</td>
<td>3.46</td>
<td>1.10</td>
<td>1.22</td>
</tr>
<tr>
<td>C10 I trust my management not to make inappropriate use of software code metrics</td>
<td>8.3</td>
<td>16.7</td>
<td>37.5</td>
<td>20.8</td>
<td>16.7</td>
<td>3.21</td>
<td>1.18</td>
<td>1.39</td>
</tr>
</tbody>
</table>

The results to questions C1 and C2 reveal that the current respondents are more likely to be using metrics to evaluate implementation than design quality. Questions C3, C4, C5 and C6 show that there is stronger support for using metrics on the respondents' own code than on the code of others. The use of metrics on code written by subordinates (question C5) is proportionately less well supported than the options associated with other classes of third-party use.
Results from Questions C7 and C9 demonstrate that metrics are also being used within projects to select areas for review, and between projects to assist with effort estimation.

The results from question C8 shows that there is a marked reluctance to use metrics in personnel evaluation decisions, while question C10 shows that respondent sentiments are broadly supportive of the notion that management is not expected to abuse metric data. It is possible that the result of question C8 is due to the low level (21%) of staff appraisal responsibilities identified by the work responsibility question. Taken together, the results of C8 and C10 might indicate that, in the respondents' work settings, there is a degree of sensitivity to the issues raised by Deming, Pagan and others relating to the use of process measurements as an input to personnel appraisal.

6.4.4 Issues relating to procedural metrics

The next group of questions relate to the gathering, interpretation, and value or otherwise of the procedural group of metrics, i.e. lines of code (LOC), McCabe's cyclomatic number (MVG), lines of comment (COM), and the ration between the first two and the third.
Table 14: Issues relating to procedural metrics

<table>
<thead>
<tr>
<th>Opinions on metrics and metrics issues (Procedural Metrics)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>MEAN</th>
<th>SD</th>
<th>VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1 I find LOC useful as an intuitive guide to the scale of a piece of work</td>
<td>4.2</td>
<td>16.7</td>
<td>9.3</td>
<td>58.3</td>
<td>12.5</td>
<td>3.58</td>
<td>1.06</td>
<td>1.12</td>
</tr>
<tr>
<td>D2 I find LOC per procedure or module useful as a predictor of inadequate decomposition</td>
<td>8.3</td>
<td>0.0</td>
<td>29.2</td>
<td>54.2</td>
<td>8.3</td>
<td>3.54</td>
<td>0.98</td>
<td>0.95</td>
</tr>
<tr>
<td>D3 I find MVG useful as a predictor of testing difficulty</td>
<td>4.2</td>
<td>4.2</td>
<td>16.7</td>
<td>75.0</td>
<td>0.0</td>
<td>3.63</td>
<td>0.77</td>
<td>0.59</td>
</tr>
<tr>
<td>D4 I find MVG per procedure or module useful as a predictor of likelihood of presence of defects</td>
<td>4.2</td>
<td>8.3</td>
<td>37.5</td>
<td>45.8</td>
<td>4.2</td>
<td>3.38</td>
<td>0.88</td>
<td>0.77</td>
</tr>
<tr>
<td>D5 I find ratios between COM and MVG useful as predictors of inadequate commenting</td>
<td>4.2</td>
<td>12.5</td>
<td>41.7</td>
<td>37.5</td>
<td>4.2</td>
<td>3.25</td>
<td>0.90</td>
<td>0.80</td>
</tr>
</tbody>
</table>

Of the statements in this group, all attracted a reasonable degree of support, with the most striking feature the 75% positive vote in favour of the value of the connection between MVG and testing difficulty. Though marginally lower, the use of LOC as a general heuristic for system sizing was also supported. MVG was also supported as a tool for predicting defect density. Of the two ratios discussed, support for the LOC/COM ratio was marginally higher than that for the MVG/COM ratio. However, given that the difference between the two means is a small fraction of the standard deviation for either question, this difference is unlikely to be significant.

6.4.5 Issues relating to structural metrics

This group of questions concerned the metrics identified in the current project as structural metrics, that is the fan-in (FI), fan-out (FO) and the derived information flow measure IF4.
Table 15: Issues relating to structural metrics

<table>
<thead>
<tr>
<th>Opinions on metrics and metrics issues (Structural Metrics)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>MEAN</th>
<th>SD</th>
<th>VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1 I have an intutitive grasp of the meaning of FI and FO</td>
<td>8.3</td>
<td>20.8</td>
<td>29.2</td>
<td>25.0</td>
<td>16.7</td>
<td>3.21</td>
<td>1.22</td>
<td>1.48</td>
</tr>
<tr>
<td>E2 I find FI a useful measure of the width of re-use of a module</td>
<td>4.2</td>
<td>16.7</td>
<td>45.8</td>
<td>33.3</td>
<td>0.0</td>
<td>3.08</td>
<td>0.33</td>
<td>0.69</td>
</tr>
<tr>
<td>E3 I have an intutitive grasp of the meaning of IF4</td>
<td>8.3</td>
<td>20.8</td>
<td>50.0</td>
<td>12.5</td>
<td>8.3</td>
<td>2.92</td>
<td>1.02</td>
<td>1.04</td>
</tr>
<tr>
<td>E4 I find IF4 a useful measure of the contribution of a module to structural risk for the overall project</td>
<td>0.0</td>
<td>4.2</td>
<td>79.2</td>
<td>12.5</td>
<td>4.2</td>
<td>3.17</td>
<td>0.56</td>
<td>0.32</td>
</tr>
<tr>
<td>E5 I believe that calculation of FI, FO and IF4 from a modules interface only gives enough information to be useful</td>
<td>0.0</td>
<td>12.5</td>
<td>79.2</td>
<td>8.3</td>
<td>0.0</td>
<td>2.96</td>
<td>0.46</td>
<td>0.22</td>
</tr>
</tbody>
</table>

The results shown in Table 15 indicate that there is little or no support among the respondents for the use of the structural metrics. The majority of respondents replied neutrally or in the negative to questions E1 and E3 relating to understanding of the definitions of the metrics. Given this background it is not surprising that questions E2 and E4 relating to the validity of these measures for a particular purpose do not attract wide support, indeed the level of support they have is perhaps more than one would expect.

The negative responses in this section are consistent with the broadly negative response to question C1 relating to use of metrics for design quality assessment, given that this group of metrics is the main provision in the current project for design-level checking.
6.4.6 Issues related to object-oriented metrics

The questions in this section relate to the metrics proposed by Chidamber and Kemerer on their articles entitled A metrics suite for object-oriented design (Chidamber & Kemerer, 1994).

Table 16: Issues relating to object-oriented metrics

<table>
<thead>
<tr>
<th>Opinions on metrics and metrics issues (Object-Oriented Metrics)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>MEAN</th>
<th>SD</th>
<th>VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1 I find WMC as useful as a measure of class complexity</td>
<td>0.0</td>
<td>12.5</td>
<td>50.0</td>
<td>33.3</td>
<td>4.2</td>
<td>3.29</td>
<td>0.75</td>
<td>0.56</td>
</tr>
<tr>
<td>F2 I find the LOC as useful as a measure of the extent of re-use of a class</td>
<td>0.0</td>
<td>4.2</td>
<td>54.2</td>
<td>41.7</td>
<td>0.0</td>
<td>3.38</td>
<td>0.58</td>
<td>0.33</td>
</tr>
<tr>
<td>F3 I find DIT useful as a measure of the difficulty of predicting behaviour of a class</td>
<td>4.2</td>
<td>4.2</td>
<td>62.5</td>
<td>29.2</td>
<td>0.0</td>
<td>3.17</td>
<td>0.70</td>
<td>0.49</td>
</tr>
<tr>
<td>F4 I find CBO useful as a measure of the coupling of a class</td>
<td>4.2</td>
<td>0.0</td>
<td>62.5</td>
<td>25.0</td>
<td>8.3</td>
<td>3.23</td>
<td>0.82</td>
<td>0.67</td>
</tr>
<tr>
<td>F5 I find RFC a useful predictor of the complexity of the behaviour of a class</td>
<td>0.0</td>
<td>0.0</td>
<td>87.5</td>
<td>12.5</td>
<td>0.0</td>
<td>3.13</td>
<td>0.34</td>
<td>0.11</td>
</tr>
</tbody>
</table>

The results revealed some support for all of the metrics proposed by these authors.

The fact that question F5 on the RFC metric had such a large neutral vote may well relate to the fact that this metric was not calculated by the CCCC tool, and hence the majority of respondents had probably not had experience of using automatically calculated values of it. This question was included because it was felt that this metric is a significant part of Chidamber and Kemerer's framework, and the fact that it was not calculated by the tool implemented as part of the current research project should not prevent respondents expressing their opinion on it.
6.4.7 Issues relating to the CCC tool

The final set of questions relate to specific features of the CCC tool implemented as part of the project. These include the novel metrics measured by the tool, and some of the report organization and presentation techniques used.
Table 17: Issues relating to features of CCCC

<table>
<thead>
<tr>
<th>Opinions on metrics and metrics issues (CCCC Features)</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>S</th>
<th>MEAN</th>
<th>SD</th>
<th>VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>G1 I have an intuitive grasp of the meaning of the IF4v measure</td>
<td>0.0</td>
<td>20.8</td>
<td>62.5</td>
<td>16.7</td>
<td>0.0</td>
<td>2.96</td>
<td>0.62</td>
<td>0.39</td>
</tr>
<tr>
<td>G2 I find IF4v a useful predictor of a module's contribution to interface integrity problems for a project</td>
<td>0.0</td>
<td>4.2</td>
<td>87.5</td>
<td>8.3</td>
<td>0.0</td>
<td>3.04</td>
<td>0.36</td>
<td>0.13</td>
</tr>
<tr>
<td>G3 I have an intuitive grasp of the meaning of the IF4c measure</td>
<td>0.0</td>
<td>25.0</td>
<td>62.5</td>
<td>12.5</td>
<td>0.0</td>
<td>2.88</td>
<td>0.61</td>
<td>0.38</td>
</tr>
<tr>
<td>G4 I find IF4c a useful predictor of a module's contribution to buildability problems for a project</td>
<td>0.0</td>
<td>4.2</td>
<td>91.7</td>
<td>4.2</td>
<td>0.0</td>
<td>3.00</td>
<td>0.29</td>
<td>0.09</td>
</tr>
<tr>
<td>G5 I find WMCv a useful predictor of a module's functional importance within a project</td>
<td>0.0</td>
<td>4.2</td>
<td>91.7</td>
<td>4.2</td>
<td>0.0</td>
<td>3.00</td>
<td>0.29</td>
<td>0.09</td>
</tr>
<tr>
<td>G6 I find colour highlighting of outlying metric values in the report generated by CCCC useful</td>
<td>0.0</td>
<td>0.0</td>
<td>54.2</td>
<td>37.5</td>
<td>8.3</td>
<td>3.54</td>
<td>0.66</td>
<td>0.43</td>
</tr>
<tr>
<td>G7 I find the ability to customise the thresholds at which colour highlighting applies useful</td>
<td>0.0</td>
<td>0.0</td>
<td>54.2</td>
<td>29.2</td>
<td>16.7</td>
<td>3.63</td>
<td>0.77</td>
<td>0.59</td>
</tr>
<tr>
<td>G8 I find the ability to generate a single report covering a project and all of its modules useful</td>
<td>0.0</td>
<td>0.0</td>
<td>45.8</td>
<td>25.0</td>
<td>29.2</td>
<td>3.83</td>
<td>0.87</td>
<td>0.75</td>
</tr>
<tr>
<td>G9 I find the ability to generate a summary report for a project with separate detailed reports on each module useful</td>
<td>0.0</td>
<td>0.0</td>
<td>50.0</td>
<td>40.8</td>
<td>29.2</td>
<td>3.99</td>
<td>0.88</td>
<td>0.78</td>
</tr>
<tr>
<td>G10 I find the ability to traverse HTML links from the tables generated by CCCC to the source code relating to the current entries in those tables useful</td>
<td>0.0</td>
<td>0.0</td>
<td>54.2</td>
<td>20.8</td>
<td>25.0</td>
<td>3.71</td>
<td>0.86</td>
<td>0.74</td>
</tr>
</tbody>
</table>

On questions G1, G2, G3 and G4 we find that there is no significant understanding of, or support for, the modified versions of the information flow metric which have been proposed in the theoretical work.
done for this project. This outcome is not surprising given the lack of support for the parent family of structural metrics discussed in section 6.4.5.

On question G5, we find that the provision of a variant WMCv, which is the weighted methods per class using visibility weighting (i.e. visible method count), also attracts no significant support.

On the other hand, the remaining questions, which relate to report generation and presentation features of the COCC tool all yield significant positive results, with in each case the mean around one standard deviation above the neutral marker.

6.4.8 Summary for agreement questions

All of the analysis above is based on the premise that the verbally defined categories in which responses were required mapped to a valid numeric scale. The calculation and reporting of figures for mean, standard deviation and variance of the distribution of responses for each question on this scale have revealed various trends among the respondents. Many of these trends were either expected for reasons external to the survey, or are found to be mutually supporting with other survey findings.

The statistical processing performed on the survey results has been deliberately kept simple. In the light of the vagueness of the meaning of the numerical agreement scale, it was not felt appropriate to apply advanced tools such as tests of significance. In formulating textual interpretations of the tables, the following rules of thumb were applied:
• Where the mean response was of the order of one standard deviation away from the neutral position, the outcome was described in terms of fairly strong support for or opposition to the proposition.

• Where the mean response was less than one quarter of one standard deviation away from the neutral position, the outcome was described in terms of no significant support being found for the proposition.

The format of the agreement questions does lend itself to investigation of second- and further-order effects using statistical techniques such as cluster analysis and analysis of variance. Consideration was given to performing some more detailed analysis of the data to find such effects, however it was decided that the sample size was not sufficient to support the use of such techniques. The raw responses to all of the factual questions (apart from the email addresses) are tabulated in Appendix F of the current thesis.

6.5 Responses to the free format question

As noted in section 5.4, the survey instrument ends with a single free-format question that enables the respondent to raise any issue he/she wishes. While some respondents did not offer any response in this area, many of them did, at some length in some cases. All of the responses received to this question are presented as Appendix G to this thesis.

6.6 A comparison survey

Some workers at the Software Metrics Research Laboratory at the University of Otago, New Zealand, have published summary tables for a survey conducted some time before mid-1997. Their research is not directly comparable to the survey in the current project, but there are
some points on which it is instructive to compare findings. The group from Otago has published this data on the Internet, in a number of HTML pages.

The set of data examines the prevalence of object-oriented development methodologies (Gray, 1997a). This survey, which must predate the one done for the current project by at least 18 months, found the following patterns:

- Of 174 respondents who identified the methodologies in use, 43 were using no methodology at all, and a little over 50% of those who were using methodologies were using more than one.

- The most popular individual methodologies were Booch, Rumbaugh (i.e. OMT), and the Unified Methodology.

While these results do not agree exactly with the findings of the current survey, the inconsistencies are easily explained in terms of the effect of the time difference at a time of rapidly changing techniques. While the survey associated with this thesis found nearly all respondents using a methodology and hardly any using more than one, the leading methodologies are the same. It is evident from examination of the SMRL group’s data that most of the respondents who reported using more than one methodology must have been reporting one of the leaders (Booch, OMT or Unified) as at least one of their choices. It is likely that the results of the Otago study are showing the market settling down, and respondents converging to one or other of the dominant groups (not forgetting that Booch and OMT are basically merging into the Unified method).
This page also contains a table reporting the percentage of respondents who report using any methodology as a function of the development language used: the percentages for languages with significant numbers of respondents (i.e., more than 10) range from 52% for Delphi up to 86% for Ada 95. The populations in this table give us the number of respondents in their survey using different languages. As with the current survey, C++ is the most popular language, in use by 124 respondents out of 184. Java is divided between two categories, Java Applications, with 30 respondents, and Java Applets with 35 respondents. The only other language with more than 30 respondents is Delphi with 48. Neither plain C nor plain Pascal are mentioned as such: perhaps these are seen as not being object-oriented languages and hence outside the scope of the investigation. Aside from the absence of C, these results are broadly in line with the findings from the current project described in section 6.3.2.

The group has also published a page with tables relating which methodologies and languages are in use relative to use of metrics (Gray, 1997b). Unfortunately the paper does not define what 'use of metrics' is in the context of this exercise, and in particular it does not identify whether the authors are talking about product metrics relating to source code analysis, or process metrics relating to tracking of productivity, defect densities, etc. The overall finding is that of 160 respondents, only 24 (15%) claim to be using metrics.

The SMRL group has also published a page reporting a statistical model which 'explains' the use of software metrics (presumably this has been
generated by regression from their data. This model is described by Gray (1997c).
Code Review Experiment

7.1 Background

The survey described in the previous chapter has some valuable attributes but is limited in a number of respects. The sample size is respectable, and the demographic questions included in the survey indicate that (assuming truthful responses) the group is representative of the target cross-section of the software engineering profession. However, the administration of the survey using the Internet gives us a response group which was self-selecting and which we would expect to be biased in favour of respondents with a degree of time on their hands. We would also expect that the respondents would be positively biased towards some aspects of code-based software metrics. The conduct of the questionnaire restricted the ability of the candidate to ensure that respondents had been exposed to the issues under investigation by use of the CCCC tool (or any similar metrics-related application) over a period of time before responding.

In the light of these issues, the survey results reported in chapter 6 (particularly those relating to the value of individual metrics) cannot be seen as an endorsement of any of the concepts the project explores. The current chapter describes an attempt to design and execute an empirical experiment in the application of the CCCC metric tool, in the hope of obtaining more conclusive evidence on the usefulness or otherwise of the specific metric techniques implemented.
7.2 The conception of the experiment

There are relatively few examples in the literature on the subject of attempts to perform empirical evaluation on the value of the use of software metrics in real industrial situations. The papers cited by Kafura and Henry (1981, 1984) are exceptions to this generalisation, as are papers by Ohlsson, Eriksson and Helander (1997) and Briand, Daly and Wust (1998). Of the work of this kind to date, the majority attempt to validate a metric as a predictor of specific quality problems by correlating it with some available attribute which can be interpreted as a proxy for the existence of the predicted problems.

While this approach has some appeal for the evaluation of metrics, it adopts a simple-minded view that the usefulness of a metric is determined solely by its ability to identify quality problems independent of human intervention. This view may not capture the subtleties of interaction between a body of software, a metrics tool and the human mind in the expected context of use. In this context, the metric analysis tool operates as a kind of decision support tool, and its value is directly related to the enhancement of a human operator's performance in the presence of the tool. The role of the tool as a component in a human-centred system is implicit in much of the material on the subject, but is considered more explicitly by Shepperd and Ince (1989).

It was decided to design and execute an experiment that attempts to gain empirical evidence to support or refute the specific assertion that access to metric-related information is capable of improving the performance of a software engineer in his/her duties. The specific work situation chosen for investigation was a source code review. The direct
way to evaluate a specific collection of metrics-related information to a specific review task would be:

- to recruit a number of subjects to perform the task;
- to group the subjects up into two categories;
- to supply only one group with the metrics-related information, and then ask both groups to complete the exercise under conditions which are otherwise identical; and,
- to compare the performance of the two groups.

It is the last step of this design outline that is the hardest part to realise. The problem is that there is no generally applicable method of evaluating the quality of the outcomes of an individual review.

Two techniques that are possible solutions to this problem are:

- choosing to perform the review exercise in such a way that the expected high quality review outcome is already known to the experimenter by selecting (or creating) an item for review which is known to have specific faults; or
- performing one or more independent reviews (perhaps using a trusted inspector) in the hope of identifying a known high-quality review outcome for comparison with the outcomes of the review under evaluation.

The first of these options might have been acceptable if the goal of the experiment had been to investigate diagnostic methods relating to specific quality risks. The review items would also need to be selected to ensure coverage of the full range of presence and absence of each such risk. Even in this case, the process of review item selection is
problematical. In the first place, items selected or created with the intent of displaying given quality issues are not necessarily representative of the gamut of items that the real-world review process has to handle. In the second place, there is a danger of bias in favour of an expected experimental outcome entering through this selection process. Delegation of the selection process to a third party independent of the experimenter can reduce, but not eliminate, the risk of such bias.

In the case of the current investigation, it was not felt appropriate to focus on a small number of specific quality issues, so the first of the two options was clearly unavailable, and a variant of the second option was employed. In addition to the 'normal' reviews conducted with and without metric information, a third group was asked to complete the same exercise in an attempt to arrive at a high-quality outcome for the exercise, as a reference against which the outcomes of the other two groups would be evaluated.

In the early stages of planning the experiment, it was considered that an attempt might be made to recruit a group of 'guru' reviewers to provide the reference group. This plan was abandoned because of perceived difficulties in identifying and securing the services of such high-quality reviewers. The final design employed a strategy that attempts to ensure a higher-quality response from the reference group. As it was not possible to guarantee that the personnel selected for the reference group were of a higher skill level than the other groups, the conditions of the experiment were manipulated to degrade the performance of the other two groups artificially. This outcome was achieved by requiring both control
and treatment groups to perform the exercise under a demanding time constraint while members of the reference group were encouraged to take as much time as necessary to achieve their optimal response.

For the remainder of this chapter, the three groups identified above will be referred to using numbers as follows:

- **Group 0**
  The reference group, charged with completing the exercise to the best of their ability with no significant time constraint or metric-related data.

- **Group 1**
  The control group, charged with completing the exercise to the best of their ability under a severe time constraint with no access to metric-related data.

- **Group 2**
  The treatment group, charged with completing the exercise to the best of their ability under an identical severe time constraint to group 1, but with access to metric-related data.

### 7.3 Design of the review exercise

The first constraining issue considered in arriving at a detailed design for the experiment was exercise performance time: there is a trade-off between time taken for a participant to complete the exercise and participation rate. Given that the current experiment had no means of providing rewards to encourage participation (or penalties to discourage non-participation), it was felt that a maximum per-subject time requirement which would not reduce participation disastrously would be of the order of one hour. This time limit (which was expressed as a
limitation on the expected time taken, rather than a compulsory time limit) was applied to group 0. As described in the section above, the other two groups were to perform under a mandatory time limit, which was set lower than this expected limit by a ratio of 4:1, i.e., 15 minutes.

The time limits for the two groups were proposed by the researcher, and confirmed by consultation with two of his work colleagues. One consideration in setting the ratio between the time allotted for group 0 and the other groups was the fact that the length of the group 0 exercise was expected to provoke a higher rate of defection. As will be discussed later in this chapter, this difference in defection rate would bring with it the risk of this group having a different (very likely lower) skill profile to the other groups. A smaller ratio between time allocations (say 2:1 instead of 4:1) would cause a risk that the higher skilled members of group 1 might deliver a set of judgements which are in fact superior to those of group 0 despite the artificial handicap of the time limit. This effect, which would be impossible to detect within the current experimental design, would confound the purpose of the experiment completely.

In order to be able to process the results of the experiment without use of subjective classification, it was evident that the response format for the exercise would have to be closed. Each respondent would be presented with a finite number of questions, each question having a pre-defined range of possible answers. Response ranges could be continuous or could use a multi-point (high, medium, low) scale. In the context of a source code review on a typical industrial project, the objective would normally be to establish an agenda for corrective action. This observation led to
the decision that the most appropriate scale to use would be a simple Boolean one, with a positive response corresponding to a requirement for action in relation to a specific review item. It was decided that the review sample would consist of 5 samples of source code, each to be a single C++ or Java class, and the same 5 quality issues would be questioned for each item, giving a total of 25 questions to be answered in each response.

Having established the time constraints and the nature of the questions to be asked, the next decision was the selection of some samples of source code to be inspected. A group at the University of Bournemouth had recently published a paper describing an experiment with a rather different methodology but similar goal (Kirsopp, Shepperd and Webster, 1999). The authors of that paper were approached to request access to the review items used in their experiment, in the hope that the results of the earlier experiment might usefully be compared with those of the current experiment. The Bournemouth researchers were kind enough to supply both their review items and the raw data arising from their experiment.

Finally, for each of the three groups, a definitive set of materials was created. The three sets of materials were composed as follows:

- **Group D**
  
  A cover page including:
  
  * generic instructions (the aim of the exercise, where to send the returns),
  * group specific instructions (i.e., for the subject complete the exercise to the best of his/her ability and take as long as required), and
  * a blank table for the boolean responses.
Following the cover page, 5 separate source code samples, each a single Java class, each starting on a new page, total length 17 pages.

* Group 1

A cover page including:

* generic instructions (the aim of the exercise, where to send the returns),
* group specific instructions (i.e., for the subject complete the exercise within a strict 15 minute time limit), and
* a blank table for the boolean responses.

Following the cover page, 5 separate source code samples, each a single Java class, each starting on a new page, total length 17 pages.

* Group 2

A cover page including:

* generic instructions (the aim of the exercise, where to send the returns),
* group specific instructions (i.e., for the subject complete the exercise within a strict 15 minute time limit), and
* a blank table for the boolean responses.

Following the cover page, 5 separate source code samples, each a single Java class, each preceded by a metrics report on that class, each starting on a new page, total length 29 pages.

The materials for each group were published as Word 97 .doc files and also generated as PostScript files. Selected parts of the materials for each group are presented in Appendix H.

As a result of feedback from early participants, the cover pages for each of the three groups underwent minor changes to the generic instructions to encompass the possibility of return of responses via email (the original text requested responses be returned via paper mail or facsimile).
7.4 Plan for processing of data

The final part of the experimental design that needed to be established before attempting to recruit volunteers to perform the experiment was to make a plan for processing the data which would be gathered. The returned data in their raw form would report the positive or negative responses of each volunteer to each of the twenty five boolean questions.

The first processing required would be to use the group 0 responses only to derive a set of nominal 'correct' responses to each of the twenty five questions. This derivation was to be done by selecting a threshold for the number of group 0 respondents who must respond positively to a question for the correct response to that question to be classified 'true'. Selection of an appropriate value for this threshold would be a matter of experimental choice. In the current experiment a policy was made that the threshold would be set at that level which gave the proportion of questions classified as 'true', and which reflected as closely as possible the proportion of positive responses across all group 0 responses to all questions.

Given the derived set of correct responses, each individual response to each question would be classified into one of four categories: true positive (TP), false positive (FP), true negative (TN), or false negative (FN). Based on these categorisations, the data would then be analysed and presented using the Receiver Operating Characteristic methodology.

Finally, the distribution of population between these four categories between groups 1 and 2 would be examined for statistical evidence of difference using the chi-squared test.
In section 7.3 above, it was reported that the materials selected for this experiment had previously been used for an experiment of similar intent at the University of Bournemouth, described by Kirsopp et al. (1999). It was hoped that reuse of these materials would lead to the possibility of useful comparisons between the two datasets. An informal examination of the two datasets was made, but due to data gathering methodology differences between the two studies (open response vs. closed response), it was decided that there was no straightforward means of performing or presenting a systematic comparison of the experimental outcomes.

7.5 Recruitment of subjects

An important feature in the planning of an experiment such as the current one is the establishment of a target number of participants. Obviously, the more subjects that participate in the exercise, the better the chances that the experiment will yield meaningful results. There are in fact two separate conditions that must be met for data derived from an experiment conducted under the current design to yield significant new knowledge:

- Firstly, the responses of the reference group, group 0, must be sufficiently coherent that we can derive some kind of quantitative algorithm for evaluating the responses from the other two groups. Further, we must have a reasonable degree of confidence that this algorithm does in fact represent some kind of measure of conformance to good professional practice.
- Secondly, there must be sufficient responses from each of the other two groups that we would be able to detect a statistically significant
difference in this measure of conformance between them, if the standard of practice of one group actually is superior to the other.

The number of respondents needed to give a reasonable chance of these two conditions being met depends to a very large extent on the nature of the responses. The first condition in particular may depend on the nature of the items selected for review. If the review items contain a small number of widely recognizable defects which map easily onto the boolean response format (for example, a four hundred line function with no comments), we may expect a high degree of coherence within the reference group, whereby, for example, 80% of respondents might agree on 80% of the individual binary responses. At the opposite extreme, we might have a selection of review items with no clear defects at all. In the best of all possible worlds, we would hope to see such a group of items greeted with a coherent set of responses with very few items identified as positive risk. However, it is possible that we would see each respondent allocating a few items to the positive group for risk according to his or her personal predispositions, giving a less coherent set of responses.

Specific algorithms for measuring the internal coherence of the group 0 responses and the conformance of the group 1 and group 2 responses to the patterns established by group 0 will be discussed later in the current chapter. At the time the project was planned, an intuitive estimate of 30 respondents (10 per group) was proposed as a target participation level. This number of subjects would, it was hoped, give sufficient data both to demonstrate group 0 coherence and to establish a differential between the performance levels of groups 1 and 2.
Consideration was given to attempting to enlist the assistance of one or more companies in the local software industry to support the research by permitting their employees to participate in the experiment during work time. This idea was discarded on the grounds that the expected labour cost of participation would be too high for private sector enterprises to absorb. There was also discussion of the possibility of staging the review exercise using student subjects as a part of a relevant course unit. This alternative was not pursued on the grounds that evidence that the metrics techniques helped students achieve better review outcomes would not imply that the same techniques are of value to experienced software engineering professionals. Given the target number of respondents, it was decided that the experiment, like the survey described in the previous chapter, should be run over the Internet in order to maximize the possible number of respondents available for participation.

As with the survey described in the previous chapter, the subjects who made up the three groups involved in the experiment were recruited using a web page on a server at Edith Cowan University. The page described the experimental design in general terms, and contained a link to an HTML form backed by a CGI script, which allowed a potential volunteer to participate by entering his/her email address. The descriptive page was publicised using the USENET news groups comp.software.measurement, comp.software-eng, comp.lang.c++ and comp.lang.java. Articles about the study were also submitted to a number of programming-related portal sites. The experiment was also promoted by releasing versions of the CCC software with links to the descriptive web page for the experiment appearing as part of the generated report.
The CGI script which backed the volunteer form was set up to automate the assignment of volunteers to one of the three groups and the sending out of the appropriate group-specific set of materials for the review exercise. It was expected that there would be a substantial defection rate of volunteers who did not complete and return the exercise. It was also expected that the group 0 volunteers would defect at a higher rate than the other two groups due to the fact of the group 0 exercise taking a longer period of time. Two policies were put in place to ameliorate the effects of this higher defection rate of group 0:

- The algorithm that assigned volunteers to groups was skewed in favour of group 0, assigning 50% of volunteers to group 0 and only 25% to each of groups 1 and 2.
- The assignment algorithm was based on a simple hash of the volunteer’s email address, so that volunteers assigned to group 0 could not achieve reassignment to a different group by simply re-volunteering (unless they did so under a different email address).

The aim of the second of these policies was not so much to keep group 0 numbers up, as to minimize discretionary transfer of volunteers from group to group based on preference for the shorter forms of the exercise, as this was seen as a potential source of experimental bias. While this policy was probably effective in preventing volunteers migrating from group to group, it could do nothing to prevent volunteers defecting from the different groups at different rates, which is also a potential source of experimental bias.

The form inviting people to volunteer for the experiment was mounted on the web site for a period of six months from February to August, 2000.
The statistics for initial registration and completed returns over this period were as shown in Table 18.

Table 18: Initial volunteer statistics

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Group 0</th>
<th>Group 1</th>
<th>Group 2</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial registration</td>
<td>20</td>
<td>9</td>
<td>6</td>
<td>35</td>
</tr>
<tr>
<td>Completed returns</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>Defection Rate</td>
<td>90.0%</td>
<td>66.7%</td>
<td>33.3%</td>
<td>74.3%</td>
</tr>
</tbody>
</table>

When the statistics above were collated, it was decided to make a final appeal to all volunteers to try to increase the number of returns, with a particular focus on attempting to increase the number of group 0 responses. A final closure date in September 2000 was set, and emails were sent out to all volunteers advising them of the extension and attaching the experimental materials appropriate to the group to which they had initially been assigned. An attempt was also made to recruit a small number of additional subjects through personal approaches to colleagues of the candidate.

After the extended return period, the statistics were as shown in Table 19.

Table 19: Final volunteer statistics

<table>
<thead>
<tr>
<th>Quantity</th>
<th>Group 0</th>
<th>Group 1</th>
<th>Group 2</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial registration</td>
<td>23</td>
<td>10</td>
<td>8</td>
<td>41</td>
</tr>
<tr>
<td>Completed returns</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>15</td>
</tr>
<tr>
<td>Defection Rate</td>
<td>73.9%</td>
<td>60%</td>
<td>37.5%</td>
<td>63.4%</td>
</tr>
</tbody>
</table>

Clearly, the total number of returns are still considerably lower than the informal targets of 10 respondents per group, and there is a danger
that the low numbers of respondents would prevent the experiment from yielding statistically significant outcomes. The wide difference in defection rates between the three groups is an issue (as foreshadowed in section 7.3), because it raises the risk that the skill level profile across the three groups may have been distorted. This distortion is a concern for two reasons:

- for the experiment to work, it is important that the group 1 and group 2 respondents should be broadly equivalent in skill level; and
- while an increase in the average skill level of group 0 would not compromise the experiment, a decrease would.

We have no way of being sure of the direction of any effect due to these distortions, but we can make some informed guesses about their causes, with a view to gauging their impact on the validity of the experiment as a whole. The most obvious explanation of the very high defection rate from group 0 is the length of the form of the exercise that this group was asked to undertake. We cannot tell whether the effect of defection was to raise or lower the average skill level of the residual volunteers in this group who did return their responses. For example:

- High-skilled mature software engineers are unlikely to be underemployed, hence these individuals might defect at a higher rate than lower skilled individuals (e.g., new graduates).
- On the other hand, completion of the exercise might be seen as a form of 'following-through', hence it might be seen as a sign of commitment and/or professionalism.

Either of the two hypothetical causal relationships described above might lead to a difference in the skill profile of the residual non-defectors
relative to the profile of the volunteers assigned to the group as a whole, and it is probably easy enough to devise any number of other similar relationships. In the present experiment we have no way of knowing whether any of these hypotheses reflect the truth, so we cannot be sure whether the skill level profile of the final group of respondents is the same as that of the other groups. This assurance is, however, not required for the experiment to be valid; we only need to be confident that the performance of group 2, working with plenty of time, will be significantly better than the other two groups, working under the time constraint. In the absence of any evidence of gross distortion of the skill profiles, we shall merely note the issue of the group 0 defection rate, and observe that our findings must be interpreted in the light of the possibility that this source of distortion may have compromised the experiment.

The difference in defection rates between groups 1 and 2 is smaller, but is also an issue. In this case there are two intuitive hypotheses regarding why volunteers assigned to group 1 volunteers were more likely to defect than those assigned to group 2:

- Volunteers were expecting to take part in an experiment relating to the use of metrics, and may have been less motivated to complete the exercise if they were assigned to group 1 and, hence, received a set of materials with no metrics information.
- The difference in defection rate between the two groups might have arisen purely by chance.

When comparing defection rates between groups 1 and 2, our requirements for the experiment not to be invalidated are more stringent than for the
comparison between the defection rate of group 0 and the other two
groups, as it is important that groups 1 and 2 have similar skill
profiles. Of the two hypotheses listed above, only the first presents a
danger of distortion of the experimental results. If there were a
relationship between skill level and interest in metrics issues, then the
level of difference observed might be enough to confound the experiment.
In the absence of such evidence in the current software engineering
literature, this factor becomes a further cautionary issue in
interpreting the experimental results, but does not of itself invalidate
the experiment.

It would have been interesting to survey the defectors and find out their
reasons for non-completion of the experiment. This action was
considered, but in view of the fact that these people had already
received one initial email and two follow-ups relating to the project it
was decided that to do so would not be consistent with good Internet
etiquette.

7.6 Statistical terminology and conventions

From the time the current experiment was conceived, it was expected that
the odds were against the amount of data gathered being sufficient to
provide clear statistical evidence relating to the usefulness of the
techniques under consideration. For this reason, it was decided to treat
the exercise as a pilot study, with the focus being on validation and
refinement of the experiment itself. One of the key objectives of the
experiment was to discover which, if any of the available statistical
processing methods would be likely to work best in comparable situations
in the future.
In statistical terms, the experiment may be characterised as an attempt to identify differences in performance between two populations (reviewers with access to metrics and those without) based on measurements of the performance of samples from those populations over a sample of work. In general, the focus of the statistical techniques we use will be to enable a binary decision between two propositions:

- the proposition that the level of differences observed between the samples of the two populations is consistent with the normal operation of chance; and
- the proposition that the level of differences between the two samples is not consistent with the operation of random effects and that there is a significant difference in the nature of the underlying populations.

The first of these propositions is often referred to as the 'null hypothesis' \( H_0 \), while the second proposition is generally referred to as the 'alternative hypothesis' \( H_A \). By convention, statistical tests are generally expressed in terms of acceptance or rejection of the null hypothesis.

Drawing an inference from statistical data is an inherently uncertain activity. Whichever of the two propositions above is inferred (and the rejection of one is the confirmation of the other), there is a possibility that an error will have been made. If the alternative hypothesis is inferred from analysis of data when the null hypothesis is actually true, this is said to be an alpha error (or type one error). If the null hypothesis is inferred when the alternative hypothesis is true, this is described as a beta error (or type two error). These are errors
only in the sense of the inference being incorrect: the diagnosis of an alpha or beta error presupposes that the experimental data was correctly gathered and processed, the incorrect inference having arisen out of the data sampled being by chance a misleading sample. In general, the focus of statistical techniques is to ensure that the probability of an alpha error is limited to a known small value. It is conventional to use either 0.01 or 0.05 as the upper bound on the acceptable level of alpha error. Throughout the current experiment, the threshold of 0.05 will be used, implying that an inference rejecting the null hypothesis will only be made when there is a probability of 0.05 or less that the null hypothesis is in fact true.
7.7 Raw data

Table 20 shows the raw data collected in the survey, with each row representing a single return. Each respondent was allocated a single-letter unique identifier, shown in the first column, with the group to which the respondent belonged being shown as the second column. The next twenty-five columns show the responses to the individual questions, with a '1' indicating a positive (risk present) response, and a '0' representing a negative response. The final column shows the time taken to complete the exercise (as reported by the particular respondent).

Table 20: Raw Data from Code Review Experiment

| Group | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 Mins |
|-------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-------|
| A     | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 50    |
| B     | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 19    |
| C     | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 43    |
| D     | 0  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 0  | 1  | 0  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 59    |
| E     | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 32    |
| F     | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 19    |
| G     | 1  | 1  | 1  | 1  | 0  | 0  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 19    |
| H     | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 18    |
| I     | 1  | 0  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 12    |
| J     | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 15    |
| K     | 2  | 0  | 0  | 1  | 0  | 0  | 1  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 15    |
| L     | 2  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 14    |
| M     | 2  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 15    |
| N     | 2  | 1  | 0  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 15    |
| O     | 2  | 0  | 0  | 1  | 1  | 0  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 0  | 1  | 1  | 0  | 0  | 15    |

At this point, we can verify that the timing data reported by respondents is consistent with our expectations arising out of the experimental design. For the design to be valid, we need to be confident that the time constraint can be relied upon to have ensured that the aggregate of group 0's judgement represents a more considered view of the exercise than that of group 1. There are two circumstances under which
examination of the timing data above would tell us that this was not true:

- If the time constraint for groups 1 and 2 were too loose, we would expect a significant proportion of group 0 volunteers to complete the exercise within the time constraint applied to groups 1 and 2.

- If the time constraint for group 0 were too tight, we would expect the majority of group 0 volunteers to use the whole time budget. Such an observation would indicate that the time constraint on group 0 had lead to some degradation relative to that group's peak possible performance. We would have no way of knowing for sure whether this degradation was only slight, or whether the time allocated was seriously inadequate so that the responses of the group contained too much noise to be used as a valid benchmark for relative evaluation of the other two groups.

We can see that all but one of the group 1 and group 2 respondents reported using the entire allotted 15 minutes plus or minus one minute. We can also see that of the group 0 respondents, who were permitted but not required to take up to an hour, the majority took at least three quarters of an hour and only one completed the exercise in a time comparable with the group 1 and group 2 respondents. In short, there is nothing in the distribution of times reported which would cause us to worry that the exercise selected for the experiment was grossly over-length or under-length for the amount of time allotted to the different groups.
Table 21 shows, for each group, the number of respondents and the count of positive responses from that group to each question.

Table 21: Cumulated Responses of Each Group

| Group | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|-------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0     | 6 | 2 | 2 | 6 | 0 | 1 | 0 | 3 | 4 | 2 | 0 | 5 | 0 | 2 | 5 | 3 | 0 | 2 | 4 | 0 | 2 | 0 | 2 | 1 | 1 |
| 1     | 4 | 1 | 1 | 3 | 2 | 2 | 0 | 1 | 3 | 2 | 1 | 3 | 2 | 2 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 1 | 1 | 2 | 1 |
| 2     | 5 | 3 | 0 | 4 | 1 | 2 | 0 | 0 | 3 | 1 | 1 | 5 | 2 | 1 | 2 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 2 | 3 | 0 |

7.8 Derivation of correct responses

The worksheet presented as Table 28: Derivation of Correct Responses in Appendix I: Statistical Worksheets shows the calculation via which a set of correct responses to the twenty-five questions in the exercise was derived, using the procedure described in section 7.4 (the worksheet itself includes further descriptive text on the procedure). The derived correct values are displayed as Table 22.

Table 22: Derived Correct Responses

<table>
<thead>
<tr>
<th>Q</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCR(0)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
7.9 Categorization of responses

Given the derived correct responses to each question, Table 23 shows the categorization of the response of each respondent to each question as true positive (TP), true negative (TN), false positive (FP) or false negative (FN).

Table 23: Categorization of Individual Responses

<table>
<thead>
<tr>
<th>Id</th>
<th>Gp</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
<th>20</th>
<th>21</th>
<th>22</th>
<th>23</th>
<th>24</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>FN</td>
<td>FP</td>
<td>TN</td>
<td>FN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>FN</td>
<td>FP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>FP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>FP</td>
<td>FP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>FN</td>
<td>TP</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>FP</td>
<td>FN</td>
<td>TN</td>
<td>TN</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>FN</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>FP</td>
<td>FN</td>
<td>TN</td>
<td>TN</td>
<td>FN</td>
<td>FP</td>
<td>TN</td>
<td>FN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TP</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
</tr>
<tr>
<td>J</td>
<td>1</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TP</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
</tr>
<tr>
<td>K</td>
<td>2</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
</tr>
<tr>
<td>L</td>
<td>2</td>
<td>FP</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
<td>TN</td>
<td>FP</td>
</tr>
<tr>
<td>O</td>
<td>2</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TP</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
</tr>
<tr>
<td>P</td>
<td>2</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TP</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
<td>TN</td>
<td>TP</td>
</tr>
</tbody>
</table>

The number of responses in each category for each group can be calculated as shown in Table 24. As well as the number of responses in each of the TP/TN/FP/FN categories, the table shows the total number of correct question responses (equal to TP+TN), the true negative fraction (TNF=TN/(TN+FP)) and the true positive fraction (TPF=TP/(TP-FN)).

Table 24: Categorization of Group Responses

<table>
<thead>
<tr>
<th>Gp</th>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>Correct</th>
<th>INF</th>
<th>TPF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30</td>
<td>89</td>
<td>18</td>
<td>12</td>
<td>119</td>
<td>.52</td>
<td>.71</td>
</tr>
<tr>
<td>1</td>
<td>13</td>
<td>54</td>
<td>16</td>
<td>15</td>
<td>67</td>
<td>.78</td>
<td>.46</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>73</td>
<td>17</td>
<td>20</td>
<td>88</td>
<td>.91</td>
<td>.43</td>
</tr>
</tbody>
</table>
7.10 Receiver Operating Characteristic Analysis

The current investigation seeks to devise a way of assessing the value of a metrics tool in a human centred system that performs useful software engineering work. The role of the tool in this system has much in common with the role assigned to diagnostic technologies such as medical imaging or blood chemistry tests in some clinical settings. Literature on this area was researched to find evaluation methodologies that might potentially be applicable to the software engineering setting.

One technique which is commonly used to describe the performance of clinical tests is Receiver Operating Characteristic analysis (Zweig and Campbell, 1993), henceforth referred to as ROC analysis. ROC analysis is a technique for evaluating the effectiveness of an ordinal scale measure as a predictor for a binary state of nature. The advantage of the ROC approach over alternatives is that there is no need to select a single decision threshold on the ordinal scale as a basis for the prediction, as the prediction calculation is performed repeatedly with different levels of the threshold. This approach ensures that all of the variability in the distribution of the ordinal predictor variable is used in the assessment of predictive performance, rather than just variability near whatever threshold is selected. The predictor variable is sometimes referred to as the decision variable (Hanley, 1989, 310): the scale of the variable may be continuous, for example as the concentration of a particular chemical in a sample of blood; or, it may be a discontinuous scale, for example a five-point confidence rating as shown in Hanley's paper (ibid, 315). The scale of the decision variable must be at least ordinal in nature, that is, it is not possible to perform ROC analysis.
unless values of the decision variable can be ordered in some way. Given a set of cases, where both the state of nature and the value of the decision variable are known, a ROC curve can be plotted. This plot shows the effect of varying the decision threshold from a value below the lowest decision variable value in the distribution, to one above the highest decision variable value. Ideally, the set of decision thresholds for which points are plotted should include at least one value between each non-equal pair of values represented in the decision variable distribution. For each threshold, the prediction performance of the decision variable relative to the selected threshold should be classified as true positive, true negative, false positive, or false negative; and, from these classifications the true positive fraction (TPF) and true negative fraction (TNF) for that threshold should be calculated and plotted. This plot gives an intuitive visual description of the performance of the decision variable as a predictor of the state of nature.

The value of the threshold which is below the lowest represented decision variable value should give rise to a point at TPF=1.0, TNF=1.0, while the value above the highest represented decision variable value should give rise to a point at TPF=0.0, TNF=0.0. As the threshold is increased between these two extremes, both TPF and TNF should fall, giving a curve which resembles a perturbed leading diagonal of the unit square bounded by (0,0),(1,1). If the decision variable has some predictive power relative to the state of nature, the perturbation of the diagonal should give rise to an upward concave form, the better the predictive power the more extreme the concavity. If there is some threshold for which all "state-of-nature=true" cases have higher decision variable value and all "state-of-nature=false" cases have lower decision variable value, the
curve will run straight up the left side and along the top of the unit square. Note that this condition can be detected even if the threshold between the two state-of-nature levels is not known when the plot is started. The area under the curve is widely suggested as a single-figure summary of the predictive value of the decision variable [Zweig and Campbell, 1993, 568; Hanley and McNeil, 1982].

In the case of the current experiment, within each group, the responses to each of the twenty five questions in the exercise are treated as separate cases. The number of respondents allocating the question to the positive (risk presence) category being treated as the decision variable, and the derived correct responses arrived at by the procedure described in section 7.8 as the state of nature. ROC analysis was performed and curves plotted for each of the three groups, although the analysis for group 0 is less meaningful than those for the other two groups due to the fact that the state of nature is actually derived from the decision variable in this case. Appendix I includes the three worksheets which present this calculation for each of the three groups. These worksheets also show the calculation of the area under the curve for each group, calculated by the simple method of summation of the areas of the set of trapezia bounded by the x-axis, the two perpendiculars rising from the x-axis to two adjacent points on the curve and the line between those two adjacent points.
Figure 2: ROC curves for groups 0, 1 and 2

Visual inspection of the curves in Figure 2 shows that group 2 appears to offer better predictive outcomes for higher values in the TPF/TNF range. Group 1 has slight superiority toward the bottom of each range, but both the range of TPF/TNF values where group 1 dominates and the differential at those values are small compared to the range and differential for the area where group 2 dominates. These visual impressions are confirmed by the calculated values of the areas under the two curves which are 0.53 for group 1 and 0.62 for group 2.
The curve also shows a curve drawn using the ROC algorithm for the responses of group 0 against the correct answers, which were derived from aggregating those responses. As the state of nature is not independent of the predictive variable in this case, this curve cannot strictly be described as an instance of the ROC technique; however, the curve has some intuitive interest and is presented for this reason. Had the individual responses of this group been a perfect predictor of the state of nature (i.e., unanimously in favour of or against each of the 25 propositions tested), we would expect the ROC to run all the way up the left-hand axis then straight along the top of the unit square, giving an area under the curve of 1.0. The shape of the curve and the area under the curve value of 0.77 are intuitively a measure of the degree of consensus within the group. It is not possible to draw major inferences from a single instance of this figure. It should be noted that an area under the curve figure close to 0.5 (e.g., in the range 0.5 to 0.55) would imply a group whose individual responses relate to the derived 'correct' figures only marginally better than a group who answered the question by tossing a coin. Clearly, the aggregated opinions of such a group would not be a good basis for establishment of the state of nature in an experiment such as the current one.

7.11 Chi-squared analysis

While the ROC analysis described shows that we have observed a difference in predictive behaviour between the control and treatment groups in our study, it does not give us any direct indication of whether this difference is evidence of systematic differences between the attributes of the two populations involved. The inferential test we used to test for such a difference is the standard chi-squared test of contingency.
The two by four contingency table for the comparison of performance of group 1 and group 2 is a subset of Table 24, and is shown on Table 25.

Table 25: Contingency table for groups 1 and 2

<table>
<thead>
<tr>
<th>Gp</th>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>13</td>
<td>54</td>
<td>18</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>73</td>
<td>17</td>
<td>20</td>
</tr>
</tbody>
</table>

The worksheet presented as Table 32 on page 265 in Appendix I shows the calculation of the chi-squared value for the contingency table arising out of the tabulation of the four kinds of responses for each of the two groups.

As the worksheet shows, the value of the chi-squared calculation is 0.9623, well below the critical level 7.8200, which would be required to claim that the experiment had demonstrated a difference in behaviour sufficiently marked that we would expect it to occur by chance on less than 5% of occasions. On this basis we report that there was no significant difference between the performance of the two groups. We conclude that the use of the metric information has not been shown to be of any value in the current case.

The chi-squared test indicates quite firmly that the current experiment cannot be claimed to provide evidence of an effect on the performance of the exercise due to the provision of metric information. There is, however, one feature of the data shown on the worksheet that is worth mentioning in case it proves useful to future experimenters in the...
current area. While the differences observed were, overall, not sufficiently strong to be statistically significant given the data available, examination of the column in the worksheet showing the difference between observed and expected results for each cell in the contingency matrix shows a striking pattern. Relatively large differences between expected and observed counts were found in all four cells relating to true negative and false positive counts for each group, with the observed counts for true positive and false negative being very close to those expected. The contributions of each cell in the contingency table to the chi-squared value is shown in Table 26.

Table 26: Chi-squared contributions for each contingency

<table>
<thead>
<tr>
<th></th>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>.0248</td>
<td>.0039</td>
<td>.3349</td>
<td>.0198</td>
</tr>
<tr>
<td>2</td>
<td>.3188</td>
<td>.0077</td>
<td>.3073</td>
<td>.0169</td>
</tr>
</tbody>
</table>

The pattern observed may be due to random variations in the data. However, future experimenters may wish to consider the possibility that the pattern is a weak signal to the effect that metrics information may have helped respondents avoid giving false positive responses, more than it helped them to give true positive responses. This view, if accepted, would tend to encourage additional experimentation specifically geared to test the role of metrics information in the false positive/true negative (FP/TN) decision context, rather than in the general context. In practical terms, a decision to bias an experiment to investigate the FP/TN decision would lead to the selection of code samples believed to be of high quality, so that the number of responses in the FP/TN categories would be maximized.
7.12 Additional investigations

Following the calculation of the chi-squared statistic for the current experiment described in the previous section, the experimental data was reviewed, and a small number of additional investigations were performed.

One additional investigation was an examination of the sensitivity of the results to selection of a threshold for the derivation of correct responses from the group 0 data. The policy described in section 7.8 for selection of the threshold is certainly defensible, however there is no overwhelming reason not to allow any convenient threshold to be selected. Of the other possible thresholds, it was found that the threshold value of 1.5 led to very similar shaped ROC curves and figures for area under the curve, but gave rise to a slightly higher chi-squared statistic of 1.75. While this is still well below the critical level of 7.82 required for significance at the 5% level, the higher value indicates a reduced probability of the differences between the two groups being due to pure chance.

A further investigation was based on the observation that the twenty five questions in each individual response do in fact display a form of fine structure. They can be organized as five groups of five responses relating to the different attributes of the same source code sample, and as five groups of five responses relating to the same attribute of five source code samples. These two ways of grouping the responses in groups of five give rise to 10 potential additional smaller datasets, which might conceivably show effects which are cancelled out in the larger dataset. A worksheet was prepared which showed the TP/FP/TN/FN performance over each of the groups of five questions relating to a
single sample or a single quality issue. This worksheet allowed contingency tables for each of these data subsets to be analysed using the chi-squared test and each of the ten subsets was analysed in this way. None of the data subsets were found to reveal performance differences between the two groups significant at the 5% level.

7.13 Summary of the experiment

To summarize the outcomes of the current experiment, we can make the statement:

No significant difference in the performance of the code review exercise was found between the control group (group 1, operating without metrics information) and the treatment group (group 2, operating with metrics information). We therefore conclude that there is no evidence from the current experiment to suggest that the metrics information is of benefit in the setting simulated by this experiment.

While the experiment failed to demonstrate a significant difference between the performance of the two groups, nonetheless it is possible to say that a difference in performance was observed. Although the statistical analysis of the data shows that the difference observed was well within the range of outcomes that might arise out of the operation of random effects, it is possible that a similar experiment with a larger number of participants might demonstrate a significant effect.
8 Conclusion

8.1 Findings in relation to the initial research questions

In section 1.2 of this thesis, the primary research questions for the project are posed. These questions are repeated below, and findings in relation to each are presented.

8.1.1 Feasibility of implementation

- Can we implement tools to measure the selected metrics in an economical way, and gather data using those tools without negative impacts on the development process?

The project has demonstrated the specification and implementation of an automated tool for gathering metrics data on source code. The development of the CCCC analyzer was conducted with the expectation that the primary use made of the tool will be by a software engineer who wishes to analyze his/her own code, although its use in other contexts is not excluded.

The development process constitutes a proof by existence of the feasibility of implementing a tool for automated analysis. Anecdotal feedback suggests that while many users are able to identify shortcomings with the program produced, they generally qualify such criticism with some statement to the effect that the CCCC program is better than no automated tool at all. Several of the comments reproduced in Appendix G are in this vein and similar sentiments were expressed in a number of email messages.
The tool was developed and promoted in such a way as to appeal to individual developers as a means to check their own work. This policy was intended to minimize the negative effects associated with some measurement programs in the absence of trust between the measurers and those who are measured. The comments referred to above, taken with the responses to survey questions, seem to indicate that this attempt was reasonably successful.

8.1.2 Theoretical validation

- Do the metrics meet the theoretical requirements for software metrics proposed in the literature?

The GQM analysis presented in chapter 3 attempts to apply a methodology for selection of measurement targets that is a widely supported framework in the recent literature of the field. This chapter also presents an analysis of some of the measures selected for implementation from the point of view of Weyuker's axiomatic approach to the value and meaning of source code based measurements. Between them, these two pieces of theoretical work demonstrate:

- that the source code attributes selected for measurement have some relationship to the quality goals of the selected project; and
- that the specific measurement algorithms chosen generate a body of data which discriminates between different samples of source code according to the extent, presence or absence of the attributes selected.

8.1.3 Practical validation

- Are the metric results of practical value in an industrial software development setting?
The responses to the questionnaire, and anecdotal evidence from the free format responses and email feedback, suggest that there is a significant body of users of the software who found the use of the analyzer had some value in the industrial setting. The lack of countervailing voices asserting that the tool was of zero value is probably due to the effect of post-evaluation apathy, but we must assume that there were some users who tried the tool out and promptly decided it held no benefit for them. The fact that the FTP logs (Appendix D) show that over 1000 users downloaded the latest version of the tool (version 2.1.4) would seem to demonstrate clearly that there is demand for such a tool, whether or not the current implementation is successful in satisfying the demand.

Interestingly, despite the wide support for the value of the tool, among the metrics implemented, the only ones for which a majority of respondents expressed positive support were LOC and MVG. The responses relating to the structural metrics were neutral or even, marginally negative, while the object-oriented metrics attracted some support without gaining a majority of positive responses on any of the questions in this area.

The code review experiment described in chapter 7 attempted to obtain empirical data to support the proposition that the metric tool was of use in a specific setting of a code review. The outcome of that experiment is that the group who had access to metrics data were not observed to have a significant benefit over the group operating without such data.
8.1.4 Interface ergonomics

- How can we design the interface through which metric tools report back results to improve the quality outcomes of the development process?

Features of the output format of the CCCC tool were strongly supported by respondents to the survey. Positive support in excess of 40% was recorded for the following features:

- colour highlighting of abnormal values;
- customization of highlighting thresholds;
- generation of separate detailed reports for individual modules bound together by hypertext links from an overall report; and
- use of hypertext links to traverse from a report datum to the source code fragment(s) which contribute to it.

The last point is particularly interesting in view of the weak implementation of the feature in the reference version of the program. In the implementation of the program which had been seen by the respondents, the links were directly to the plain text source files, and hence could not seek directly to the required line, but only to the start of the file. More recent versions of the program have overcome this problem by creating HTML copies of the source files with embedded anchors to allow direct jumping to the exact lines specified.

Although the features of the HTML report were widely appreciated, there were a number of requests for other formats, notably a flat file export format suitable for loading into a spreadsheet or database. There was also one request for an interface similar to the HTML tabular format, but with dynamically sorting columns (comparable to those in the Windows 95 Explorer interface).
8.1.5 Breadth of application

- How general are the lessons learned in this study? Can they be applied in situations involving other metrics, or to organizations which have different operational situations?

Throughout the project, the Greek/Roman cultural dichotomy introduced in section 2.10 is used as a framework for consideration of issues of workplace culture. The culture of the industry partner's project group around which the requirements were defined would be characterized as lying at the Greek end of the spectrum. The way the tool was released and promoted would also tend to favour its adoption by Greeks (who choose their own tools) rather than Romans (for whom the tools are selected by management). As noted in Chapter 6, the majority of the respondents to the survey would also be primarily of the Greek persuasion.

The conclusion we draw from this is that, to the extent to which the data gathered is of any value at all, it applies to developers and groups who meet the following profile:

- operating as intellectual free agents with the ability to select their own tools and methods (within certain limits); and
- more likely to be working on high-innovation, real-time embedded or control projects than low-innovation transaction processing, accounting or data management systems.

8.2 Other findings

In addition to the findings recorded above in relation to the research questions posed at the start of the project, there are a number of other areas where the project has uncovered new information.
8.2.1 Novel metrics

As part of the process of metric selection via the GQM analysis, a number of novel metrics were proposed in the hope of illuminating aspects of modern object-oriented programming practice.

The most significant innovations are the versions of the structural metrics filtered according to visibility (Flv, FoV and IF4v) and concreteness (Fin, FoC and IF4c). The survey contained two questions focussed on each of these modified metrics, asking firstly whether the respondent felt that he/she understood the basis of calculation of each modified metric, and secondly whether he/she felt that the metric was a useful indicator of project risks. The distribution of responses to these four questions (reported in section 6.4.7) indicates that very few of the respondents were confident about the basis of calculation or wished to express a positive or negative opinion on the value of the modified metric. In view of the fact that the underlying unmodified IF4 metric was subject to similar confusion, one cannot reject the modified metrics as being definitely devoid of value. It is, however, clear that if these metrics (or similar ones relating to the same quality attributes) are to become useful, clearer explanations of their bases must be provided than the ones which were published in the current project.

Another novel metric which was calculated by the tool, and about which a question was included in the survey was WMCv, the weighted methods per class metric, applying a weighting function which effectively filtered the visible methods from the interface. Again, there was neither endorsement nor firm rejection of the value of this metric.
8.2.2 Use of pragmatic measurement techniques

One issue that was identified early in the implementation phase for the analyzer was the trade-off between the number of items measured and the accuracy with which the measurements were done. Put simply, given a certain finite development effort to build the analyzer, the use of coarse, pragmatic counting methods allowed a greater range of metrics to be calculated than would have been possible if only rigorous techniques had been employed. The user guide for CCC (which appears in Appendix C) has a section on counting methods, which describes the main pragmatic techniques applied, with the potential sources of error inherent in each.

The survey included a question relating to the specific issue of using interface source code to calculate the structural metrics. The distribution of responses to this question was centred around the neutral point with a small bias to the negative.

From the feedback received through the free-format responses on the questionnaire and email, it appeared that the 3-5% tolerance for error in calculation methods was acceptable to most users.

8.3 Suggestions for further work

The following lines of investigation are suggested by the outcomes of the current project.

8.3.1 Further development of the analyzer

Section 4.5 identifies a number of potential enhancements to the existing analyzer that might add to its immediate value and support further research.
Among the enhancements suggested, the ones relating to extending the data model to encompass abstraction at the level of class categories would be particularly interesting to pursue. In the light of the support for the object-oriented metrics suite of Chidamber and Kemerer, it would also be worthwhile to invest some effort in attempting to add support for the two metrics LCOM and RFC which are not presently supported by CCCC.

8.3.2 Consideration of structural metrics

The survey revealed that the group of structural metrics calculated by the current analyzer are not well understood or supported. These metrics were designed to characterize some of the major risk issues identified in the industry partner's project. A project could be undertaken to investigate this area of metrics technology further. The outcomes of this investigation might be alternate metrics to fill the gap, or improved documentation to raise the levels of acceptance of the metrics currently provided.

8.3.3 Further statistical analysis of survey data

The analysis undertaken on the data gathered in the practitioner survey was limited to identifying the distribution of responses to each question. Further analysis could be undertaken to examine correlations between responses to different questions. The raw data gathered in the survey (with the identifying data in the email and free-format questions removed) will be made available on the project web site.

8.3.4 Use of the analyzer to support research in other areas

There are at least two projects presently using the CCCC analyzer to support uses of metrics other than the source code risk assessment around which the tool is designed. One group is using it to attempt to discover
design patterns in code (Kim & Boldyreff, 1997), while another is using it as a calibration tool for an investigation into retrospective calculation of an object-oriented derivative of function points from code (Judge & Williams, 1997, Judge & Mistry 1998).

8.3.5 Further empirical work

The experiment described in chapter 7 failed to uncover any significant benefit from the use of software metrics in a code review setting, however it was handicapped by a level of participation lower than that which was originally conceived. The experimental design and processing methods appear to be sound, and it would be interesting to repeat the experiment in the hope that a higher level of participation might allow stronger conclusions to be drawn. In the event that this proves possible, either or both of the following amendments to the current experimental design should be considered:

- While the current experiment rejected the use of student volunteers on the grounds that they would be unrepresentative of professional software engineering behaviour, on reflection it might be acceptable to use students as the group 1 and 2 population provided that the group 0 response was drawn from experienced software professionals, and providing that some effort was made to match the levels of competence of the groups of students assigned to group 1 and group 2.

- Although the data arising out of the current experiment were not classified as significant, there is a pattern in that data suggestive of a hypothesis that metrics data may be more useful in avoiding false positive risk judgements than in achieving true positive ones. A future investigator might choose to pursue this hypothesis. Alternately, an investigator might choose to select materials to
favour the investigation of the true negative/false positive decision behaviour as this might lead to an overall effect which is easier to detect.
Appendix A: Specification for the prototype analyzer

The primary aim of the initial implementation of the metric analyzer will be to gather and present information on the attributes of source code to the programmers charged with the generation and/or maintenance of that code. A secondary goal will be to create a tool capable of assisting in retrospective analysis of bodies of code by investigators who have no intention of modifying or developing that code.

The analyzer will measure and report values for lines of code, comments, the McCabe cyclomatic number, and Rajaraman and Lyu's C++ coupling metrics.

The analyzer design will provide a modular framework for the implementation of multiple attribute measurements on one or more source files by invoking a single command from the operating system command processor interface. The design will be expected to localise dependencies on source language, host operating system, and output formats (as well as the range of metrics actually to be calculated), so that each of these can be substituted with minimal disruption to the others.

The analyzer will accept command-line qualifiers to control the range of metrics to be presented, output formats, etc. Invocation will appear to be a single-pass process to the user, and the elapsed time for an exhaustive analysis of a single piece of code should be considerably less
than the elapsed time required by the compiler in use to perform a normal
compilation of that code.

Informal target: analysis time $\leq (\text{compile time} \times 0.2)$

The initial implementation will parse the language C++ (as defined by the
reference manual for AT & T Unix System Laboratories C++ version 3.0). By extension, this will be usable to parse the vast majority of programs
in ANSI C (the exceptions being C programs that contain C++ reserved
words as identifiers). The implementation may provide additional logic
to detect and parse correctly these exceptional programs, or programs
written in older versions of C, or other programs which for one reason or
another are not completely valid programs in C++.

The initial implementation will operate on a generic Unix platform. Versions for MS/DOS and OS/2 should also be produced if at all possible.

The program will provide a modest range of output options selected from
the following:

- output of a single-screen summary report to an ANSI terminal using
colour to emphasize suggestive outcomes (possibly green-yellow-red to
show nominal, abnormal and dangerous values);
- output of a tabular report in columns to a device or file, especially
where multiple source files are examined;
- output of a diagramatic summary of the structure and attributes of a
piece of source code to a hard copy device;
- insertion of a report into a source code file as a comment.
Appendix B: Sample Run Input and Output

This section contains two sample C++ source code files (taken from the analyzer's own source code), the log messages on standard error generated by a run of the analyzer, and the generated HTML files for the run as a whole and one of the classes covered by the run.

The examples shown below are indicative of the function of the analyzer. The particular results of this run have not been manually checked and may be subject to some of the sources of error identified in the thesis.
Sample header file:

```c
/*
 * cccc_db.h
 * defines the database used by CCCC to generate a report
 */
#ifndef
#define CCCC_DB_H

#include <iostream.h>
#include <iomanip.h>
#include "cccc_seg.h"
#include "cccc_met.h"
#include "cccc_tbl.h"
#include "cccc_utl.h"

#ifndef
#define CCCC_DB_L

// these are generic classes, used to implement the relational model
// for the application
class CCCC_Field;
class CCCC_Record;

// these are the types of the entities in this application's relational
// model
class CCCC_Project;
class CCCC_Module;
class CCCC_Member;
class CCCC_UseRelationship;
class CCCC_Extent;
```
The entities held within the database need to be able to return a variety of kinds of name including a simple name (typically one word), a fully qualified local name (i.e. as used within a class), and a fully qualified global name.

They also need to be able to supply a key for searching, which must be
unique within the table, and a ranking string, which need not be unique
and may contain artificial elements to enforce appropriate ordering.
Subclasses may also have particular other names, which should be defined
using negative indexes:

```cpp
enum NameLevel { nLOCAL, nSIMPLE, nLOCAL, nGLOBAL
};
```
class CCCModule : public CCCRecord
{
    friend class CCCProject;
    CCCField module_name, module_type;
public:
    CCCTable<CCCMember, 1000> member_table;
    char* name(int name_level) const;
    CCCModule(istream& module_data_line);
    void generate_report(ostream&);

    // the following definitions are designed to support the Chidamber and
    // Kemerer metric suite for object oriented design
    typedef float (*MemberWeightingFunctionPtr)(CCCMember*);
    int depth_of_inheritance_tree();
    int number_of_children();
    int coupling_between_objects();
    float weighted_methods_per_class(MemberWeightingFunctionPtr);

    int get_count(const char* count_tag);
    int get_relationships(CCCProject* proj, int mask,
        CCCUseRelationship** rel_array);
    int is_trivial();
};

enum MemberNameLevel { nMEMBER_NAME=-1, nMEMBER_TYPE=-2, nMEMBER_PARAMS=-3 };

class CCCMember : public CCCRecord
{
    friend class CCCProject;
    Friend class CCCModule;
    CCCField member_type, member_name, param_list;
    CCCModule* parent;
public:
    char* name(int index) const;
    CCCMember(istream& member_data_line, CCCModule* parent_ptr=NULL);
    void generate_report(ostream&);
    int get_count(const char* count_tag);
    Visibility get_visibility();
};

enum ExtentNameLevel { nFILENAME=-1, nLINESNUMBER=-2, nDESCRIPTION=-3 };

class CCCExtent : public CCCRecord
{
    friend class CCCRecord;
    Friend class CCCProject;
    CCCField filename;
    CCCField linenum;
    CCCField description;
    CCCField flags;
    CCCField count_buffer;
    UseType ut;
    Visibility v;
public:
    CCCExtent();
    CCCExtent(istream& is);
    char* name(int index) const;
    void generate_report(ostream&);  
    Visibility get_visibility() const { return v; }
}
int get_count(const char *count_tag);
useType get_usetype() const { return ut; }
const CCCS_Strings get_description() const { return description; }
};

enum UserNamelnlevel { nSUPPLIER=-1, nCLIENT=-2, nMEMBER=-3 };
class CCCS_UsRelationship : public CCCS_record
{
  CCCS_Ficld supplier, client, member;
  useType ut;
  AugmentedBool visible, concrete;
public:
  char *name(int index) const;
  CCCS_UsRelationship(istream &is);
  void add_extent(istream &);
  int get_count(const char *count_tag);
  useType get_usetype() const { return ut; }
  AugmentedBool is_visible() const { return visible; }
  AugmentedBool is_concrete() const { return concrete; }
  void generate_report(generate CS);
  CCCS_Module* supplier_module_ptr(CCCS_Project *prj);
  CCCS_Module* client_module_ptr(CCCS_Project *prj);
};

#endif // CCCS_DB_H
Sample implementation file:

```c
/*
 * cccc_db.cc
 */

#include <sstream.h>
#include <fstream.h>
#include "cccc_db.h"
#include "cccc.h"
#define LINE_BUFFER_SIZE 1000
extern CCC_Project *prj;

// the file scope variable last_supplier is used to suppress repeated
// output of the supplier name in the use relationship section where
// the current record has the same supplier as the previous one
// the indentation makes this reasonably clear
static CCC_String last_supplier="";

// when we add a record to, for example, the extent table for a member of
// a module, we need to merge the information in the new extent
// with what is already known
// there are two kinds of merge:
// 1. ordinary fields like module_type should either be consistent or
// blank for all extents relating to the same module, so where the old
// field is blank, we overwrite with the new field
// 2. the flags field in CCC_Member contains a variety of single character
// flags giving the visibility, constness, etc. of the member, with '0'
// used to reflect a state of lack of knowledge; in these cases,
// void Resolve_Fields(CCC_Field& field1, CCC_Field& field2)
// if(strlen(field1)--0)
// {field1=field2;}
```
```c
void CCCC_Record::merge_flags(CCCC.Fields new_flags)
{
    char *new_flag_array=new_flags;
    char *flag_array=flags;
    unsigned int len=strlen(flag_array);
    if (strlen(new_flag_array)>len)
    {
        char buf[10];
        unsigned int i;
        for (i=0; i<len; i++)
        {
            if (flag_array[i]=='?')
            {
                buf[i]=new_flag_array[i];
            }
            else
            {
                buf[i]=flag_array[i];
            }
        }
        buf[len] = '\0';
        flags=buf;
    }
    else
    {
        // if the parent record has just been created it may have
        // an empty flags member, so we use Resolve.Fields to copy
        // the flags from the first extent
        Resolve.Fields(flags, new_flags);
    }
}

void CCCC_Record::add_extent(istream &is)
{
    CCCC.Extent *new_extent=new CCCC.Extent(is);
    CCCC.Extent *inserted_extent=extent_table.find_or_insert(new_extent);
    merge_flags(inserted_extent->flags);
    if (new_extent != inserted_extent)
    {
        delete new_extent;
    }
}

char* CCCC_Record::name(int /* level */ const { return ""; } char* CCCC_Record::ranking_string() const { return name(nlRank); } char* CCCC_Record::key() const { return name(nlRank); } void CCCC_Record::generate_report(ostream &os)
{
    os << name(nlRank) << endl;
}

int rank_by_string(const void *pl, const void *p2) {
    CCCC_Record *pr1=*(CCCRecord**p1);
    CCCC_Record *pr2=*(CCCRecord**p2);
    // we collect and use local copies of the two strings
    // this allows the ranking string functions to use static buffers
    CCCC_String s1=pr1->ranking_string();
    CCCC_String s2=pr2->ranking_string();
    return strcmp(s1, s2);
}
```
char* CCCC_Module::name(const char* name_level) const
{
    static CCCC_String retval;
    switch(name_level)
    {
    case n|MODULE_TYPE:
        retval.module_type;
        break;
    case n|MODULE_NAME:
        retval.module_name;
        break;
    case n|MODULE_TYPE_AND_NAME:
        retval.module_type;
        if(strlen(retval)==0)
        {
            retval="-";
        }
        retval=retval.module_name;
        break;
    default:
        retval.module_name;
        if(strlen(retval)==0)
        {
            retval="Anonymous";
        }
    }
    return retval;
}

int CCCC_Record::get_count(const char* /*count_tag */) {
    // CCCC_Record is really an abstract base class, so we don't expect
    // anyone to call this method
    // if they do, warn them off...
    cerr << "[CCC_Record::get_count] virtual function, please overload" << endl;
    return 0;
}
void CCCC_Project::assign_anonymous_members()
{
    cerr << "Assigning non-member functions to source modules" << endl;
    int i;
    for(i=0; i<member_table.records(); i++)
    {
        CCCC_Member *member_ptr=member_table.record_ptr(i);
        CCCC_String member_name=member_ptr->name(nlSEARCH);
        CCCC_String parent_name=member_ptr->parent->name(nlMODULE_NAME);
        if(strlen(parent_name)==0) {
            II search the extent list for the best parent
            II our rule for assigning parentage to source files is that it will be
            II the first seen definition of the function, or if no definitions
            II are seen the first seen declaration
            II providing that this function is run before the database is sorted,
            II the records should be in order of creation
            II we can assume that at least one extent has been seen
            CCCC_Extent *best_extent_ptr=member_ptr->extent_table.record_ptr(0);
            int j;
            for(j=1; j<member_ptr->extent_table.records(); j++)
            {
                // if the candidate we already have is a definition, we should
                // never replace it
                if(best_extent_ptr->ut == utDEFINITION)
                    break;
            }
            else if(member_ptr->extent_table.record_ptr(j)->ut == utDEFINITION)
            {
                best_extent_ptr=member_ptr->extent_table.record_ptr(j);
            }
            CCCC_String source_module_name=best_extent_ptr->name(nlFILENAME);
            cerr << "Assigning " << member_name
                 << " to source module " << source_module_name << endl;
            MAKE_STRSTREAM(str);
            str << source_module_name << "Source file" << ends;
            CCCC_Module *new_mod_ptr= new CCCC_Module(str);
            CCCC_Module *lookup_mod_ptr=module_table.find_or_insert(new_mod_ptr);
            if(lookup_mod_ptr != new_mod_ptr)
            {
                delete new_mod_ptr;
            }
            member_ptr->parent=lookup_mod_ptr;
            RELEASE_STRSTREAM(str);
        }
    }
}
void CCCC_Project::sort() {
    int i;

    cerr << "Sorting modules" << endl;
    module_table.sort();
    for (i = 0; i < module_table.records(); i++)
        module_table.record_ptr(i)->sort();

    cerr << "Sorting members" << endl;
    member_table.sort();
    for (i = 0; i < member_table.records(); i++)
        CCCC_Member *member_ptr = member_table.record_ptr(i);
        member_ptr->sort();
        member_ptr->parent->member_table.append(member_ptr);

    cerr << "Sorting relationships" << endl;
    userel_table.sort();
    for (i = 0; i < userel_table.records(); i++)
        userel_table.record_ptr(i)->sort();
}

CCCC_Project::CCCC_Project(const CCCC_String& name)
    : project_name(name)
{
    // we prime the database with knowledge of the builtin base types
    // we also add a record for the anonymous class which we will treat
    // as the parent of all non-member functions
    char *builtin_type_info[] = {
        "void@builtin@<nofile>@O@builtin definition@d????@@0d",
        "int@builtin@<nofile>@O@builtin definition@d????@@0d",
        "char@builtin@<nofile>@O@builtin definition@d????@@0d",
        "long@builtin@<nofile>@O@builtin definition@d????@@0d",
        "float@builtin@<nofile>@O@builtin definition@d????@@0d",
        "double@builtin@<nofile>@O@builtin definition@d????@@0d",
        NULL
    };
    for (char **ptr = builtin_type_info; *ptr != NULL; ptr++)
    {
        MAKE_STRSTREAM(type_info);
        type_info << *ptr << ends;
        add_module(CONVERT_STRSTREAM(type_info));
        RELEASE_STRSTREAM(type_info);
    }
}
void CCCC_Project::add_module(istream module_data_line) 
{ 
    CCCC_Module *module_ptr=new CCCC_Module(module_data_line); 
    CCCC_Extent *extent_ptr=new CCCC_Extent(module_data_line); 
    CCCC_Module *lookup_module_ptr=module_table.find_or_insert(module_ptr); 
    if(lookup_module_ptr != NULL) 
    { 
        lookup_module_ptr->extent_table.find_or_insert(extent_ptr); 
    } 
    if(lookup_module_ptr->module_ptr) { 
        // do some work to transfer knowledge from the new module object 
        // then delete it 
        Resolve_Fields(lookup_module_ptr->module_type,module_ptr->module_type); 
        delete module_ptr; 
    } 
} 

void CCCC_Project::add_member(istream& member_data_line) 
{ 
    CCCC_Module *module_ptr=new CCCC_Module(member_data_line); 
    CCCC_Module *lookup_module_ptr=module_table.find_or_insert(module_ptr); 
    CCCC_Member *member_ptr= 
        new CCCC_Member(member_data_line,lookup_module_ptr); 
    CCCC_Member *lookup_member_ptr=member_table.find_or_insert(member_ptr); 
    if(lookup_member_ptr != NULL) 
    { 
        lookup_member_ptr->add_extent(member_data_line); 
    } 
    // if the member pointer we created here is not exactly equal to the one 
    // found in the table, it has not been stored in the table, either because 
    // the table already contained the a pointer to the same member, or because 
    // there was no space 
    // either way, the one we just created is an orphan, so we delete it 
    if(member_ptr != lookup_member_ptr) 
    { 
        delete member_ptr; 
    } 
} 

void CCCC_Project::add_userel(istream& userel_data_line) 
{ 
    CCCC_UserRelationship *new_userel_ptr = 
    new CCCC_UserRelationship(userel_data_line); // 
    CCCC_UserRelationship *lookup_userel_ptr= 
    userel_table.find_or_insert(new_userel_ptr); 
    if(lookup_userel_ptr != NULL) 
    { 
        if(new_userel_ptr != lookup_userel_ptr) 
        { 
            delete new_userel_ptr; 
        } 
        lookup_userel_ptr->add_extent(userel_data_line); 
    } 
} 

void CCCC_Project::add_rejected_extent(istream& rejected_data_line) 
{ 
    CCCC_Extent *new_extent=new CCCC_Extent(rejected_data_line); 
    rejected_extent_table.find_or_insert(new_extent); 
}
float unit_weighting(CCCC_Member *mpt) { return 1.0; }

float visible_weighting(CCCC_Member *mpt) { float retval;
Visibility mvis=mpt->get_visibility();
switch (mvis) {
case vPRIVATE:
   retval=0.0;
   break;

case vPROTECTED:
case vPUBLIC:
case vUNKNOWN:
   retval=1.0;
   break;

default:
   cout << "Unexpected visibility " << (int) mvis << endl;
   retval=0.5;
}
return retval;
}

int CCCC_Project::get_count(const char* count_tag) {
   // we could give this method knowledge of which tags are likely
   // to be found in which table, ... or we could be lazy like this
   int retval=0;
   retval+=module_table.get_count(count_tag);
   retval+=userel_table.get_count(count_tag);
   return retval;
}

void CCCC_Project::generate_report(ostream& os) {
   os << "CCCC Report" << endl << flush;
   sort();
   int i;
   for (i=0; i<module_table.records(); i++) {
      CCCC_Module *mptr=module_table.record_ptr(i);
      if (strcmp(mptr->name, "builtin") != 0)
         mptr->generate_report(os);
   }
   for (i=0; i<userel_table.records(); i++) {
      CCCC_UserRelationship *uptr=userel_table.record_ptr(i);
      uptr->generate_report(os);
   }
}

CCCC_Module::CCCC_Module(istream& module_data_line) {
   module_data_line >> module_name >> module_type;
}
void CCCC_Module::generate_report(ostream & os) {
    if(strlen(module_name) > 0) {
        os << module_type << " " << module_name << endl;
    } else {
        os << "<non-member functions and variables>" << endl;
    }
    for(i=0; i<extent_table.records(); i++) {
        extent_table.record_ptr(i)->generate_report(os);
    }
    os << endl;
    os << "Methods:" << endl;
    for(i=0; i<member_table.records(); i++) {
        member_table.record_ptr(i)->generate_report(os);
    }
    os << endl;
    os << "\f" << endl;
}
int CCCC_Module::get_count(const char* count_tag) {
    int retval=0;
    CCCC_Relationship* dummy_peers;

    if (strcmp(count_tag,"NOH")==0) {
        retval=tri_trivial();
    } else if(strcmp(count_tag,"DIM")==0) {
        retval=depth_inheritance_tree();
    } else if(strcmp(count_tag,"PCC")==0) {
        retval=number_of_children();
    } else if(strcmp(count_tag,"CBO")==0) {
        retval=coupling_between_objects();
    } else if(strcmp(count_tag,"OMC")==0) {
        retval=(int)weighted_methods_per_class(unit_weighting);
    } else if(strcmp(count_tag,"WMC")==0) {
        retval=(int)weighted_methods_per_class(visible_weighting);
    } else if(strcmp(count_tag,"MEMV")==0) {
        retval=(int)weighted_methods_per_class(visible_weighting);
    } else if((strcmp(count_tag,"T") == 0) ||
              (strcmp(count_tag,"PO") == 0))
    {
        int role;
        int vis=meHIDDEN_OR_VISIBLE;
        int con=meABSTRACT OR_CONCRETE;

        switch(count_tag[1])
        {
            case 'O':
                role=meSUPPLIER;
                break;
            case 'I':
                role=meCLIENT;
                break;
        }

        switch(count_tag[2])
        {
            case 'v':
                vis=meVISIBLE;
                break;
            case 'c':
                con=meCONCRETE;
                break;
            case 0:
                // no qualifier
                break;
            default:
                cerr << "Unexpected FI/FO qualifier" << endl;
                return 0;
        }
    }

    return retval;
}
```c
retval = get_relationships(prj, role|vin|con, dummy peers);
else if (strcmp(count_tag, "IF1", 1) == 0)
{
    char fitag[5], ftag[5];
    const char *qualifier = &count_tag[11];
    strcpy(fitag, "FI");
    strcat(fitag, qualifier);
    strcpy(ftag, "PO");
    strcat(ftag, qualifier);
    int fanout = get_count(fitag);
    int fanin = get_count(ftag);
    retval = fanout * fanout * fanin * fanin;
}
else
{
    retval = member_table.get_count(count_tag);
}
return retval;

int CCC Module::depth_of_Inheritance_tree()
{
    int retval = 0;
    CCC_USERrelationship **supplier_relationships;
    int num_suppliers = get_relationships(prj);
    for(int i = 0; i < num_suppliers; i++)
    {
        CCC_USERrelationship *this_rel = supplier_relationships[i];
        if (this_rel->get_usertype() == INHERITS)
        {
            int dit_this_path = this_rel->supplier_module_ptr(prj) . depth_of_Inheritance_tree() + 1;
            if (dit_this_path > retval)
            {
                retval = dit_this_path;
            }
        }
    }
    return retval;
}
```
int CCCC_Module::number_of_children() {
    int retval = 0;
    CCCC_UseRelationship **supplier_relationships;
    int num_suppliers = get_relationships(
        prj, rmeCLIENT|rmeHIDDEN_OR_VISIBLE|rmeABSTRACT_OR_CONCRETE, supplier_relationships);
    for(int i = 0; i < num_suppliers; i++) {
        CCCC_UseRelationship *this_rel = supplier_relationships[i];
        if (this_rel->get_usertype() == utINHERITS) {
            retval++;
        }
    }
    return retval;
}

int CCCC_Module::coupling_between_objects() {
    // Coupling between objects (CBO) is defined as the total number
    // of distinct modules to which the current module has relationships
    // we calculate this by identifying client and supplier relationship
    // numbers, and then subtracting the number of modules which appear in
    // both categories.
    CCCC_UseRelationship *lcl_suppliers[100], *lcl_clients[100], **relarray;
    int num_suppliers = get_relationships(
        prj, rmeCLIENT|rmeHIDDEN_OR_VISIBLE|rmeABSTRACT_OR_CONCRETE, relarray);
    memcpy(lcl_suppliers, relarray, num_suppliers * sizeof(CCCC_UseRelationship*));
    int num_clients = get_relationships(
        prj, rmeSUPPLIER|rmeHIDDEN_OR_VISIBLE|rmeABSTRACT_OR_CONCRETE, relarray);
    memcpy(lcl_clients, relarray, num_clients * sizeof(CCCC_UseRelationship*));
    int duplicate_modules = 0;
    for(int i = 0; i < num_clients; i++) {
        for(int j = 0; j < num_suppliers; j++) {
            if (lcl_clients[i]->client_module_ptr(prj) == lcl_suppliers[j]->supplier_module_ptr(prj)) {
                duplicate_modules++;
            }
        }
    }
    return num_clients + num_suppliers - duplicate_modules;
}
float CCCC_Module::weighted_methods_per_class(MemberWeightingFunctionPtr wfptr)
{
    float retval=0;
    for(int i=0; i<member_table.records(); i++)
    {
        retval+=(*wfptr)(member_table.record_ptr(i));
    }
    return retval;
}
CCCMember::CCCMember(istream& member_data_line, CCCCModule *parent_ptr) {
    parent = parent_ptr;
    member_data_line >> member_name >> member_type >> param_list;
}

char *CCCMember::name(int name_level) const {
    static CCCC_String retval;

    MAKE_STRSTREAM(namestr);
    switch(name_level) {
    case nlRANK:
    case nlSEARCH:
        // there is no scoping for C-style functions ... 
        if(
            (strcmp(parent->name(nlMODULE_NAME),"\0")!=0) &&
            (strcmp(parent->name(nlMODULE_TYPE),"file")!=0) 
        )
        {
            namestr << parent->name(nlMODULE_NAME) << ":;";
        }
    namestr << member_name << param_list << ends;
    break;
    case nlMEMBER_NAME:
    case nlSIMPLE:
        namestr << member_name << ends;
    break;
    case nlMEMBER_TYPE:
        namestr << member_type << ends;
    break;
    case nlMEMBER_PARAMS:
        namestr << param_list << ends;
    break;
    case nlLOCAL:
        namestr << member_type << ":" << member_name << param_list << ends;
    break;
    default:
        cerr << "unexpected name level" << endl;
    }
    retval=namestr.str();
    RELEASE_STRSTREAM(namestr);
    return retval;
}

void CCCMember::generate_report(ostream& os) {
    os << endl << setw(65) << name(nlLOCAL) << flags << endl;
    int i;
    for(i=0; i<extent_table.records(); i++) {
        extent_table.record_ptr(i)->generate_report(os);
    }
}
int CCCC_Member::get_count(const char* count_tag) {
    int retval = extent_table.get_count(count_tag);
    return retval;
}

Visibility CCCC_Member::get_visibility() {
    Visibility retval = vDONTRNOK;
    for (int i = 0; i < extent_table.records(); i++) {
        Visibility visibility_for_this_extent =
            extent_table.record_ptr(i)->get_visibility();
        // in theory, all extents which define visibility should define it
        // similarly: in practice, we don't want to assume this, so we assume
        // that the first extent we process which defines visibility is accepted
        // as authoritative
        if (retval == vDONTRNOK) &&
            (visibility_for_this_extent != vDONTRNOK)
        {
            retval = visibility_for_this_extent;
            break;
        }
    }
    return retval;
}

void CCCC_Extent::generate_report(ostream& os) {
    os.setf(ios::right, ios::right | ios::left);
    os << setw(20) << filename;
    os << setw(20) << linenumber;
    os.setf(ios::left, ios::left | ios::right);
    os << setw(20) << description;
    os << flags << " " << count_buffer << endl;
}

CCC_Extent::CCC_Extent() {
    // we can trust the CCCC_String constructor to give us empty strings,
    // but we need to initialise these
    v = vDONTRNOK;
    ut = utDONTRNOK;
}

CCC_Extent::CCC_Extent(istream& is) {
    is >> filename;
    if (!is.eof())
    {
        is >> linenumber >> description >> flags >> count_buffer
            >> v >> ut;
    }
    else
    {
        // we can trust the CCCC_String constructor to give us empty strings,
        // but we need to initialise these
        v = vDONTRNOK;
        ut = utDONTRNOK;
    }
}
char *CCCC_Extent::name(int name_level) const
{
    static CCC_String retval;
    MAKE_STRSTREAM(namestr);
    switch(name_level)
    {
    case nlFILENAME:
        namestr << filename << ends;
        break;
    case nlLINENUMBER:
        namestr << linenumber << ends;
        break;
    case nlDESCRIPTION:
        namestr << description << ends;
        break;
    case nlRANK:
    case nlLOCAL:
        namestr << flags << " " << filename << " " << linenumber
               << " " << get_description() << ends;
        break;
    case nlSIMPLE:
        namestr << filename << ":" << linenumber << ends;
        break;
    default:
        cerr << "unexpected name level" << endl;
    }
    retval=namestr.str();
    RELEASE_STRSTREAM(namestr);
    return retval;
}

int CCCC_Extent::get_count(const char* count_tag) {
    int retval=0;
    char local_count_buffer[100], *count_tag_ptr, *count_value_ptr;
    strcpy(local_count_buffer,count_buffer);
    count_tag_ptr=strtok(local_count_buffer,":") ;
    while(count_tag_ptr!=NULL) {
        count_value_ptr=strtok(NULL," ");
        if(strcmp(count_tag_ptr, count_tag)==0) {
            retval+=atoi(count_value_ptr);
        }
        count_tag_ptr=strtok(NULL,":");
    }
    return retval;
}
UseRelationship::UseRelationship(istream& is)
{
    is >> client >> member >> supplier;
    visible=abDONTKNOW;
    concrete=abDONTKNOW;
    ut=utDONTKNOW;
}

char *UseRelationship::name(int name_level) const
{
    static cccc_string retval;
    MAKE_STRSTREAM(namestr);
    switch(name_level)
    {
    case nlRANK:
        case nlSIMPLE:
            namestr << client << " uses " << supplier << ends;
            break;
    case nlSUPPLIER:
        namestr << supplier << ends;
        break;
    case nlCLIENT:
        namestr << client << ends;
        break;
    default:
        cerr << "unexpected name level" << endl;
    }
    retval=namestr.str();
    RELEASE_STRSTREAM(namestr);
    return retval;
}
void CCCC_Relationship::add_tracked_extent(Utility aUti) {
    // processing is the same as the "add_extent" method, except that we update
    // the database with corresponding table members
    // this is a "case" statement
    CCCC_Extent *new_extent = new Extent(aUti);
    CCCC_Extent *inserted_extent = find_or_insert(new_extent);

    // "new" extent isn't NULL

    case USE:
        case PARTITION:
            visible = TRUE
            break;
    case PRIVATE:
        case IMPLEMENTATION:
            visible = FALSE
            break;
    default:
        // nothing required
        
    
    // a single relationship record represents all connections between two
    // modules, hence it may have multiple extents which are of different use
    // types
    // the use type attached to the relationship record is used only to identify
    // inheritance relationships
    UseType new_ut = new_extent->get_usetype();
    if (new_ut == UT_INHERITS) {
        ut = UT_INHERITS;
    }
    switch (new_ut) {
        case UT_INHERITS:
        case UT_HASSVAL:
        case UT_PAREVAL:
        case UT_VARYVAL:
            concrete = aTRUE;
            break;
        default:
            // no change required
            
    }
    if (new_extent != inserted_extent) {
        delete new_extent;
    }
}

void CCCC_Relationship::generate_report(ostream &os) {
    os << setw(20) << client << " uses " << setw(10) << supplier
        " member " << " flags " " endl;
    for (int i=0; i<extent_table.records(); i++) {
        os << extent_table.record_ptr[i]->name(nLOCAL) « endl;
    }
}
int CCCC_Module::use_relationship::get_count(const char* count_tag) {
    return extent_table.get_count(count_tag);
}

CCCC_Module* CCCC_Module::use_relationship::supplier_module_ptr(CCCC_Project *prj) {
    CCCC_Module* retval=0;
    int i;
    for(i=0; i<prj->module_table.records(); i++)
    {
        CCCC_String mod_name =
            prj->module_table.record_ptr(i)->name(MODULE_NAME);
        if(strcmp(mod_name, supplier) == 0)
        {
            retval=prj->module_table.record_ptr(i);
            break;
        }
    }
    return retval;
}

CCCC_Module* CCCC_Module::use_relationship::client_module_ptr(CCCC_Project *prj) {
    CCCC_Module* retval=0;
    int i;
    for(i=0; i<prj->module_table.records(); i++)
    {
        CCCC_String mod_name =
            prj->module_table.record_ptr(i)->name(MODULE_NAME);
        if(strcmp(mod_name, client) == 0)
        {
            retval=prj->module_table.record_ptr(i);
            break;
        }
    }
    return retval;
}
```c
int CCCC_Module::in_trivial()
{
    int retval = 1;

    if (strcmp(module_type, "builtin") == 0)
        retval = 1;
    else if (member_table.records() > 0 ||
             strcmp(module_type, "class") == 0 ||
             strcmp(module_type, "C++ class") == 0 ||
             strcmp(module_type, "namespace") == 0 ||
             strcmp(module_type, "Java class") == 0 ||
             strcmp(module_type, "Java interface") == 0 ||
             strcmp(module_type, "Ada package") == 0)
    {
        // these are the expected module types which are always non-trivial
        retval = 0;
    }
    else
    {
        // we've never heard of it, and it doesn't have any members
        // treat it as trivial
        retval = 1;
    }
    return retval;
}
```
```c
int CCCC_Module::get_relationships()
    CCCC_Project *prj, int mask, CCCC_UseRelationship **rel_array)
{
    int number_of_relationships=0;
    const int max_relationships=100;

    static CCCC_UseRelationship *rel_ptr[max_relationships];
    memset(rel_ptr, sizeof(rel_ptr), 0);

    int i;
    for(i=0; i<prj->userel_table.records(); i++)
    {
        CCCC_UseRelationship *userel_ptr = prj->userel_table.record_ptr(i);
        CCCC_String compare_name;
        CCCC_String peer_name;
        switch (mask & trueCLIENT | trueSUPPLIER)
        {
        case trueCLIENT:
            compare_name=userel_ptr->name(n1SUPPLIER);
            peer_name=userel_ptr->name(n1CLIENT);
            break;
        case trueSUPPLIER:
            compare_name=userel_ptr->name(n1CLIENT);
            peer_name=userel_ptr->name(n1SUPPLIER);
            break;
        default:
            cerr << "[get_relationships] did not specify CLIENT or SUPPLIER" << endl;
            return 0;
        }

        if(!(strcmp(compare_name, this->name(n1SIMPLE))==0) &
            (strcmp(compare_name, peer_name)==0))
        {
            CCCC_Module *supplier_ptr=userel_ptr->supplie_module_ptr(prj);
            CCCC_Module *client_ptr=userel_ptr->client_module_ptr(prj);

            int nontrivial_flag=
            [supplier_ptr != NULL] &
            [supplier_ptr->is_trivial()] &
            [client_ptr != NULL] &
            [client_ptr->is_trivial()];

            // we don't give the benefit of the doubt on visibility;
            // if we have seen a declaration scoped as private, we
            // exclude this item from the visible count, otherwise we
            // include it
            int visibility_flag;
            if(userel_ptr->is_visible()==abFALSE) {
                visibility_flag=mask&trueHIDDEN;
            } else {
                visibility_flag=mask&trueVISIBLE;
            }

            int concrete_abstract_flag;
            if(userel_ptr->is_concrete()==abTRUE) {
```
concrete_abstract_flag=mask&concrete(CONCRETE;
}
else
{
concrete_abstract_flag=mask&ABSTRACT;
}

if(nontrivial_flag && visibility_flag && concrete_abstract_flag)
{
// a suitable case for treatment...
rel_ptrs[number_of_relationshs]*=rel_ptr;
number_of_relationshs++;
}

rel_array=rel_ptrs;
return number_of_relationshs;
}
Standard error output from analyzer run

CCCC - a code counter for C and C++

A program to analyse C and C++ source code and report on
some simple software metrics
Version 2.1.4
with contributions from Bill McLean, Herman Hueni, Lynn Wilson and Peter Bell.

The development of this program was heavily dependent on
the Purdue Compiler Construction Tool Set (PCCS)
by Terence Farr, Bill Cohen, Hank Diets, Russel Quong,
Tom Koop and others.

This software is provided with NO WARRANT

Processing cccc.db.h as ANSI C/C++
Processing cccc.db.cc as ANSI C/C++

Generating HTML reports

Sorting modules
Sorting members
Sorting relationships
Generating report on CCCC_Extent to .\ccc.htm\CCCC_Extent.htm
Generating report on CCCC_Field to .\ccc.htm\CCCC_Field.htm
Generating report on CCCC_Html_Stream to .\ccc.htm\CCCC_Html_Stream.htm
Generating report on CCCC_Member to .\ccc.htm\CCCC_Member.htm
Generating report on CCCC_Module to .\ccc.htm\CCCC_Module.htm
Generating report on CCCC_Project to .\ccc.htm\CCCC_Project.htm
Generating report on CCCC_Record to .\ccc.htm\CCCC_Record.htm
Generating report on CCCC_UseRelationship to .\ccc.htm\CCCC_UseRelationship.htm
Generating report on anonymous to .\ccc.htm\anonymous.htm
67 lines appended from library file cccc_inf.dat

Primary HTML output is in cccc.htm
## Project Summary

This table shows measures over the project as a whole.

- **NOM = Number of modules**
  - Number of non-trivial modules identified by the analyser. Non-trivial modules include all classes, and any other module for which member functions are identified.

- **LOC = Lines of Code**
  - Number of non-blank, non-comment lines of source code counted by the analyser.

- **COM = Lines of Comments**
  - Number of lines of comment identified by the analyser.

- **MVG = McCabe's Cyclomatic Complexity**
  - A measure of the decision complexity of the functions which make up the program. The strict definition of this measure is that it is the number of linearly independent routes through a directed acyclic graph which maps the flow of control of a subprogram. The analyser counts this by recording the number of distinct decision outcomes contained within each function, which yields a good approximation to the formally defined version of the measure.

- **L_C = Lines of code per line of comment**
  - Indicates density of comments with respect to textual size of program.

- **M_C = Cyclomatic Complexity per line of comment**
  - Indicates density of comments with respect to logical complexity of program.

- **JF4 = Information Flow measure**
  - Measure of information flow between modules suggested by Henry and Kafura. The analyser makes an approximate count of this by counting inter-module couplings identified in the module interfaces.
Object Oriented Design

- **WMC** = Weighted methods per class
  The sum of a weighting function over the functions of the module. Two different weighting functions are applied: WMC1 uses the nominal weight of $1$ for each function, and hence measures the number of functions, WMCv uses a weighting function which is $1$ for functions accessible to other modules, $0$ for private functions.

- **DIT** = Depth of inheritance tree
  The length of the longest path of inheritance ending at the current module. The deeper the inheritance tree for a module, the harder it may be to predict its behaviour. On the other hand, increasing depth gives the potential of greater reuse by the current module of behaviour defined for ancestor classes.

- **NOC** = Number of children
  The number of modules which inherit directly from the current module. Moderate values of this measure indicate scope for reuse, however high values may indicate an inappropriate abstraction in the design.

- **CBO** = Coupling between objects
  The number of other modules which are coupled to the current module either as a client or a supplier. Excessive coupling indicates weakness of module encapsulation and may inhibit reuse.

The label cell for each row in this table provides a link to the module summary table in the detailed report for the module in question.
Structural Metrics Summary

- **FI** = Fan-in
  - The number of other modules which pass information into the current module.
- **FO** = Fan-out
  - The number of other modules into which the current module passes information.
- **IF4** = Information Flow measure
  - A composite measure of structural complexity, calculated as the square of the product of the fan-in and fan-out of a single module. Proposed by Henry and Kafura.

Note that the fan-in and fan-out are calculated by examining the interface of each module. As noted above, three variants of each of these measures are presented: a count restricted to the part of the interface which is externally visible, a count which only includes relationships which imply the client module needs to be recompiled if the supplier's implementation changes, and an inclusive count. The label cell for each row in this table provides a link to the relationships table in the detailed report for the module in question.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Fan-in</th>
<th>Fan-out</th>
<th>Fan-in</th>
<th>Fan-out</th>
<th>IP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCC.C_Extent</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CCC.C_Field</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>CCC.C_Html_Stream</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CCC.C_Member</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CCC.C_Project</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CCC.C_Record</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CCC.User Relationship</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>anonymous</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Procedural Metrics Summary

For descriptions of each of these metrics see the information preceding the project summary table. The label cell for each row in this table provides a link to the functions table in the detailed report for the module in question.

<table>
<thead>
<tr>
<th>Module Name</th>
<th>LOC</th>
<th>MVG</th>
<th>COM</th>
<th>L-C</th>
<th>M-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>CCC.C_Extent</td>
<td>69</td>
<td>11</td>
<td>4</td>
<td>17250</td>
<td>2.750</td>
</tr>
<tr>
<td>CCC.C_Field</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>CCC.C_Html_Stream</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>CCC.C_Member</td>
<td>62</td>
<td>16</td>
<td>3</td>
<td>13400</td>
<td>3200</td>
</tr>
<tr>
<td>CCC.C_Project</td>
<td>295</td>
<td>65</td>
<td>14</td>
<td>21071</td>
<td>4714</td>
</tr>
<tr>
<td>CCC.C_Record</td>
<td>179</td>
<td>25</td>
<td>22</td>
<td>8136</td>
<td>1136</td>
</tr>
<tr>
<td>CCC.User Relationship</td>
<td>46</td>
<td>8</td>
<td>8</td>
<td>5750</td>
<td>1000</td>
</tr>
<tr>
<td>anonymous</td>
<td>117</td>
<td>25</td>
<td>10</td>
<td>11706</td>
<td>2500</td>
</tr>
</tbody>
</table>

Rejected Extents

<table>
<thead>
<tr>
<th>Location</th>
<th>Text</th>
<th>LOC</th>
<th>COM</th>
<th>MVG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No extents were rejected in this run.
About CCCC

The program CCCC is made available freely and with NO WARRANTY in the hope that it may be useful. The complete source code of the current version of CCCC is available from the CCCC home page at http://www.fslr.ac.ewan.edu.au/~tlittle.
The CCCC users guide is available at http://www.fslr.ac.ewan.edu.au/~tlittle/ccc_cccc_uh.htm. If you are on a machine where CCCC is installed, there may also be a local copy at one of these locations:

* /usr/local/lib/ccc/ccc_uh.htm (on Unix-like systems)
* c/ccc_uh.htm (on DOS/Windows systems)

The development of CCCC has been performed to support research for the degree of MSc at Edith Cowan University by Tim Littlefair.

Users of CCCC are encouraged to mail Tim Littlefair or visit the CCCC home page, and also to encourage their friends to use CCCC.

The development of CCCC was heavily dependent on an excellent tool called PCCTS, the Purdue Compiler Construction Toolset, by Terence Parr, Will Cohen, Hank Dietz, Russell Quong and others, and now maintained by Tom Moog. Thanks also to Hermann Hucal, Bill McLean and Lynn Wilson for their contributions to CCCC development.
Detailed report on module

**CCCC_Member**


<table>
<thead>
<tr>
<th>Metric</th>
<th>Tag</th>
<th>Overall</th>
<th>Per Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>LOC</td>
<td>67</td>
<td>13</td>
</tr>
<tr>
<td>McCabe's Cyclomatic Number</td>
<td>MVG</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>Lines of Comment</td>
<td>COM</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>LOC/COM</td>
<td>L C</td>
<td>13.406</td>
<td></td>
</tr>
<tr>
<td>MVG/COM</td>
<td>M C</td>
<td>3.206</td>
<td></td>
</tr>
<tr>
<td>Information Flow measure (inclusion)</td>
<td>IF4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Information Flow measure (visible)</td>
<td>IF4v</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Information Flow measure (concrete)</td>
<td>IF4c</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Weighted Methods per Class (weighting = unity)</td>
<td>WMC1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Weighted Methods per Class (weighting = visible)</td>
<td>WMCv</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Depth Inheritance Tree</td>
<td>DIT</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Number of Children</td>
<td>NOC</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Coupling between objects</td>
<td>CBO</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**Relationships**

<table>
<thead>
<tr>
<th>Clients</th>
<th>Suppliers</th>
</tr>
</thead>
<tbody>
<tr>
<td>[CV]</td>
<td>CCCC_Module [CV]</td>
</tr>
<tr>
<td>cccc_db.cc:312</td>
<td>cccc_db.cc:314</td>
</tr>
</tbody>
</table>
### Functions

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>LOC</th>
<th>MVC</th>
<th>COM</th>
<th>REL</th>
<th>MTC</th>
</tr>
</thead>
</table>
| `CCCC_Member( istream& ,  
CCCC_Module* )`            | 5   | 0   | 0   |     |     |
| `cccc db.cc:567`           |     |     |     |     |     |
| `void generate_report( ostream& )` | 6   | 1   | 0   |     |     |
| `cccc db.cc:619`           |     |     |     |     |     |
| `int get_count( char* )`   | 3   | 1   | 0   |     |     |
| `cccc db.cc:627`           |     |     |     |     |     |
| `Visibility get_visibility( )` | 17  | 5   | 4   |     | 1.250 |
| `cccc db.cc:632`           |     |     |     |     |     |
| `char* name( int )`        | 30  | 9   | 1   |     | 9.000 |
| `cccc db.cc:573`           |     |     |     |     |     |
Appendix C: User Guide for CCCC

User Guide for CCCC

Table of Contents

Introduction

Report Contents

Counting Methods

Command line syntax

Configuration

Getting CCCC

Introduction

CCCC is a tool for the analysis of source code in various languages (primarily C++), which generates a report in HTML format on various measurements of the code processed. Although the tool was originally implemented to process C++ and ANSI C, facilities have recently been added to allow Java and Ada 95 source files to be recognized and processed as well. The name CCCC stands for 'C and C++ Code Counter'. Measurements of source code of this kind are generally referred to as 'software metrics', or more precisely 'software product metrics' (as the term 'software metrics' also covers measurements of the software process, which are called software process metrics). There is a reasonable consensus among modern opinion leaders in the software engineering field that measurement of some kind is probably a Good Thing, although there is less consensus on what is worth measuring and what the measurements mean.

CCCC has been developed as freeware, and is released in source code form, although precompiled binaries for Linux and Windows are included in the distribution for the convenience of users. Users are encouraged to compile the program themselves, and to modify the source to reflect their preferences and interests. The simplest way of using CCCC is just to run it with the names of a selection of files on the command line like this:

```
ecce my_types.h big.h small.h *.cc
```

CCCC will open each of the files specified on the command line (using standard wildcard processing if appropriate), and parse it using a parser selected to match the filename extension. As the parser processes each file, recognition of certain constructs will cause records to be written into an internal database. When all files have been processed, a report on the contents of the internal database will be generated in HTML format. By default the HTML report is generated to the file cccc.htm in the current working directory, although the output filename is configurable.

The report contains a number of tables identifying the modules in the files submitted and covering:

- measures of the procedural volume and complexity of each module and its functions;
- measures of the number and type of the relationships each module is a party to either as a client or a supplier;
- identification of any parts of the source code submitted which the program failed to parse; and
- a summary report over the whole body of code processed of the measures identified above.
Some of the data presented in the report may be displayed in an emphasized form (either with a bold or italic font, or with a red or yellow background). These are items which have been identified as lying outside ranges which have been laid down as desirable for the particular item. A bold font or red background indicates a value which exceeds a threshold defined as being dangerous for that measure, while italic fonts and yellow backgrounds indicate values below the danger threshold but still above a second lower threshold which has been laid down to indicate cause for concern. The two thresholds are configurable by the user of the tool; see the section below on configuring metric treatment for more details.

Report Contents

The reports generated by CCCC normally consist of the following parts:

- A project report which contains four tables:
  - project summary
  - OO design metrics
  - structural summary
  - extents rejected by the parser
- A module report per non-trivial module which contains three tables:
  - module summary
  - procedural detail
  - structural summary

Tables generated

<table>
<thead>
<tr>
<th>Table name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Summary</td>
<td>This table presents summary values of various measures over the body of source code submitted.</td>
</tr>
<tr>
<td>Object Oriented Design Metrics</td>
<td>This table presents values of some of the metrics from Chidamber and Kemerer’s Metrics Suite for Object Oriented Design.</td>
</tr>
<tr>
<td>Procedural Summary</td>
<td>This table presents values of procedural measures summed for each module identified in the code submitted.</td>
</tr>
<tr>
<td>Procedural Details</td>
<td>This table presents values of the same procedural measures covered in the procedural summary report, but this time broken down within each module into the contributions of each member function of the module. By default the procedural details table for each module is presented as part of the separate report for that module.</td>
</tr>
<tr>
<td>Structural Summary</td>
<td>This table presents counts of fan-in and fan-out relationships to each module identified, and a derived metric called the Henry/Kafura/Shepperd measure, which is calculated as the square of the product of the fan-in and fan-out counts.</td>
</tr>
<tr>
<td>Structural Details</td>
<td>This table presents lists of the modules contributing to the relationship counts reported in the structural summary. By default the lists of suppliers and clients are generated as part of the separate report for each module.</td>
</tr>
<tr>
<td>Rejected Extents</td>
<td>This table presents a list of code regions which the analyzer was unable to parse.</td>
</tr>
</tbody>
</table>
Metrics displayed

<table>
<thead>
<tr>
<th>Tag</th>
<th>Metric Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>Lines of Code</td>
<td>This metric counts the lines of non-blank, non-comment source code in a function (LOC), module (LOCm), or project (LOCp). LOC was one of the earliest metrics to come into use (principally because it is straightforward to measure). It has an obvious relation to the size or complexity of a piece of code, and can be calibrated for use in prediction of maintenance effort, although concern has been expressed that use of this metric as a measure of programmer productivity may tend to encourage verbose programming practices and discourage desirable simplification.</td>
</tr>
<tr>
<td>MVG</td>
<td>McCabe's Cyclomatic Complexity</td>
<td>A measure of a body of code based on analysis of the cyclomatic complexity of the directed acyclic graph which represents the flow of control within each function. First proposed as a measure of the minimum number of test cases to ensure all parts of each function are exercised, it is now widely accepted as a measure for the detection of code which is likely to be error-prone and/or difficult to maintain.</td>
</tr>
<tr>
<td>COM</td>
<td>Comment Lines</td>
<td>A crude measure comparable to LOC of the extent of commenting within a region of code. Not very meaningful in isolation, but sometimes used in ratio with LOC or MVG to ensure that comments are distributed proportionately to the bulk or complexity of a region of code.</td>
</tr>
<tr>
<td>LCMC</td>
<td>LOC/COM, MVG/COM</td>
<td>See above</td>
</tr>
<tr>
<td>FOM, FOM, FOM, FOM, FOM, FOM</td>
<td>Fan-out, Fan-in</td>
<td>For a given module A, the fan-out is the number of other modules which the module A uses, while the fan-in is the number of other modules which use A. See the section below on counting methods for a discussion of the distinction between the variants on each of these measures; these figures.</td>
</tr>
<tr>
<td>HKS, HKSy, HKSe</td>
<td>Henry-Katua/Shepperd measure</td>
<td>This metric is derived by squaring the product of the fan-in and fan-out of each module. The original Henry-Katua measure, which has been described as a measure of information flow complexity, includes a term for the length of the module under consideration, but CCCC uses the measure as modified by Shepperd, which omits this term on the basis that it debases the measure by combining two attributes which can and should be separately measured. Corresponding to the variants on the fan-in and fan-out measures described above, similar variants are calculated on this metric.</td>
</tr>
<tr>
<td>NOM</td>
<td>Number of modules</td>
<td>Number of modules identified in the project. See discussion below about what constitutes a module.</td>
</tr>
<tr>
<td>WMC</td>
<td>Weighted methods per class</td>
<td>This measure is a count of the number of functions defined in a module multiplied by a weighting factor. Two weighting functions are implemented, giving rise to two variants on the measure:</td>
</tr>
<tr>
<td>DIT</td>
<td>Depth of inheritance tree</td>
<td>This measure counts the maximum number of classes on any inheritance path ending at the current module.</td>
</tr>
<tr>
<td>NOC</td>
<td>Number of children</td>
<td>This measure counts the number of subclasses which inherit directly from the current module.</td>
</tr>
<tr>
<td>CBO</td>
<td>Coupling between objects</td>
<td>This measure counts the number of other modules which have client or supplier relationships to the current module.</td>
</tr>
<tr>
<td>REJ</td>
<td>Rejected lines</td>
<td>This is a measure of the number of non-blank non-comment lines of code which was not successfully analysed by the parser. This is more of a</td>
</tr>
</tbody>
</table>
Counting methods

CCCC implements simple algorithms to calculate each of the measures presented. The algorithms are intended to present a useful approximation to the underlying quantities, rather than meticulously exact counting: in general agreement with manual counts based on the same definitions should agree with CCCC to within 2-3%. Larger discrepancies are discovered, or if this level of agreement is not considered adequate, users are welcome to modify the source code to implement closer agreement, or to change the counting behaviour to reflect a desired basis of calculation. The basic definitions of each count are as follows:

- **Number of Modules (NOM)**
  CCCC defines modules in terms of a grouping of member functions: C++ classes and namespaces, Java classes and interfaces and Ada packages are all defined as modules. In addition to this, functions which are not members of one of these structures are treated as belonging to an anonymous module.

- **Lines of Code (LOC)**
  This count follows the industry standard of counting non-blank, non-comment lines of source code. Preprocessor lines are treated as blank. Class and function declarations are counted, but declarations of global data are ignored. There may be some double counting of lines in class definitions as the algorithm treats the total over a module as the sum of lines belonging to the module itself and lines belonging to its member functions (the declarations and definitions of member functions in the body of the class definition will contribute to both counts).

- **Comment Lines (COM)**
  Any line which contains any part of a comment for the language concerned is treated as a comment by CCCC, and is allocated to the module and member of the next following 'real' token. This ensures that leading comments are treated as part of the function or class definition which follows them. There is one exception to this rule: the Rational Rose design tool is known to insert directives into source files disguised as C++ comments with the prefix '//##'. These are not counted as comments, however all other comment structures are, even if there is no content within the comment delimiters, or the content is commented-out source code.

- **McCabe's Cyclomatic Complexity (MVC)**
  The formal definition of cyclomatic complexity is that it is the count of linearly independent paths through a flow of control graph derived from a program. A pragmatic approximation to this can be found by counting language keywords and operators which introduce extra decision outcomes. This can be shown to be quite accurate in most cases. In the case of C++, the count is incremented for each of the following tokens: 'if','while','for','switch','break','&&','|&|'.

  Note that the boolean operations introduce extra paths through the code because the second operand may or may not be evaluated according to the value of the first operand. Note also that the treatment of switch statements is problematic: it is quite common for multiple 'case' labels to be attached to the same block of code, so counting these might overstate the value. Counting the 'break' tokens instead is better so long as there are no case labels in the middle of the block of code which the break terminates. The motive for counting the 'switch' token is to provide for the default case, which gives rise to a path whether or not the programmer defines a default label. Counting the break token in this way may distort the count where it is used in other contexts (i.e. to exit from a block).

- **Weighted methods per class (WMC)**
  This is a count of the member functions known to exist in a class. Knowledge of existence of a function is only gained from declarations or definitions directly contained in files processed by CCCC: files included by a preprocessor are ignored, and CCCC does not at present identify invocations of member or non-member functions within procedural code.
• Fan-In, Fan-Out (FI, FIC, FIV, FO, FOC, FOV)
  Traditionally, use relationships between modules were identified by counting function invocations or access to module data in procedural code. CCC identifies relationships only through structures apparent in the definitions of the interfaces of C++ classes or Java classes or interfaces. The specific relationships which can be detected are inheritance of a supplier class by a client, containment of an instance of a supplier class in a client, and the existence of member functions of the client class which accept or return an instance of the supplier.
  While these relationships may seem unrelated to the invocation and module data counts, they are likely to show a strong correlation because of the fact that in an object-oriented environment, it is likely (but not inevitable) that the low-level use relationships of invocation and direct access to data structures require an object of the class of the supplier module to be available. This availability can be through instantiation of an instance of the supplier class within procedural code, but will often be due to the existence of one of the higher level relationships described above.
  The counts of Fan-In and Fan-Out are regarded as a measure of the structural quality of a program, with high values of either (and particularly high values of both within the same module) indicating increased risk of changes required in one module requiring changes across other modules. CCC chooses to define the relationship counts in such a way that each supplier or client module is counted only once, however many separate ways the relationship is detected. CCC applies filtering to the relationships identified to distinguish between different kinds of uses which may carry with them different levels of structural risk. There are two filters: visibility and concreteness. The visibility filter removes from consideration relationships which are known to be only accessible from the private interface of a module. Relationships which are defined in the visible part of the interface can be exploited by clients of the current module, thus forcing those clients also to be clients of the current module's supplier. Visible relationships also increase the range of operations available on an object, thus increasing the cognitive complexity of the interface from the point of view of a programmer required to use a module.
  The concreteness filter removes from consideration relationships which do not create a dependency of the implementation of the client module on the implementation of the supplier class. Dependency-creating relationships increase risk because they may not be cyclical, and thus inhibit the creation of other relationships. They also inhibit the ability of modules to be built separately, requiring recompilation of the client module when the supplier changes. The test for this filter in C++ is whether a forward declaration of the supplier class is adequate to allow the client module definition to be compiled: containment and parameter passing where the client module is modified by a reference operator are allowed in this case, containment or passing by value or inheritance are all dependency-creating. In Java, relationships except inheritance are treated as non-dependency creating.
• Number of Modules (NOM)
  All instances of the following syntactic constructs are treated as modules: C++ classes and namespaces; Java classes and interfaces. Ada packages. There are contexts where the analyser detects something which may belong to one of these categories but may not. In this case, the name is treated as a module name if and only if member functions are identified for it. Functions which do not belong to a module of one of the categories defined above are treated as belonging to a single anonymous module; if any members are identified for this module it is also counted.

Command-line syntax

The following flags control the operation of CCC:

• -h
  This option causes CCC to display a usage message and exit without processing any files.
• -r <string>
  This flag controls which of the tables of the CCC report are generated on a particular run. See the usage message output by cccc -h for up-to-date details of what reports are available.
• -d <string>
  This flag turns on debugging output for different parts of the program. <string> is a sequence of letters which turn on optional debug output, including the following:
Other debug output may be available according to the version; see ccccmain.c for the full range of flags supported.

-<filename>
This option allows a list of files to be processed to be specified in a file rather than listed on the command line

-<libdir>
This option instructs CCCC to use a specific directory for the configuration and support files used. If this option is specified the files are read from /usr/local/lib/cccc on Unix systems and c:\cccc on DOS/Windows systems.

-x <lang>
This option allows the user to specify which of the supported parsers to use on the code submitted in a single run. This is only likely to be useful when the default mapping of file extensions to languages does not correctly identify the appropriate language for a file. Supported languages are:

- C
- C++
- ada
- java
- auto

C and C++ share the same parser, although there may be different behaviour according to which is selected (either by specification using this option or due to file extension processing).

-n <project name>
This option allows the user to specify a name for the project which will be output into the header of the HTML report generated.

-o <output filename>
This option allows the user to specify an alternate name for the report generated by the analyser. The default output filename is cccc.htm.

-q
This option causes CCCC to run in quiet mode, generating less debug output.

The descriptions above are correct as of version 2.1.1. Additional functions may be available in that version or any other; please consult the source code file ccccmain.c for full details of command line handling.

Configuration

CCCC can be configured by editing various configuration files, all of which are found in the library directory described above.

Treatment of metric values

The file cccc_mnt.dat allows the user to configure the thresholds at which the HTML report presents measures in different ways. The version of this file shipped with CCCC describes the configuration data and format:

```
# cccc_mnt.dat
#
# configuration file for treatment of metric values in CCCC
#
# lines in this file starting with '!' are treated as comments
# all other lines are treated as defining a record in a table of
# treatments for different metrics, which control the display of
# values of that metric
#
# all metric values are displayed using the class CCCC_Metric, which may be
# viewed as ratio of two integers associated with a character string tag
```
# the denominator of the ratio defaults to 1, allowing simple counting to
# be handled by the same code as is used for ratios
# the tag associated with a metric is used as a key to lookup a record
# describing a policy for its display (class Metric_Treatment)
# the fields of each treatment record are as follows
# TAG the short string of characters used as the lookup key
# T1, T2 two numeric thresholds which are the lower bounds for the ratio of
# the metric's numerator and denominator beyond which the
# value is treated as high or extreme by the analyzer
# these will be displayed in emphasized font, and if the browser
# supports the BGCOLOR attribute, extreme values will have a red
# background, while high values will have a yellow background
# the intent is that high values should be treated as suspicious but
# tolerable in moderation, whereas extreme values should always
# be regarded as defects [not necessarily that you will fix them]
# NT a third threshold which suppresses calculation of ratios where
# the numerator is lower than NT
# the principal reason for doing this is to prevent ratios like L_C
# being shown as "infinity" and displayed as extreme when the
denominator is 0, providing the numerator is sufficiently low
# suitable values are probably similar to those for T1
# W the width of the metric (total number of digits)
# P the precision of the metric (digits after the decimal point)
# Comment a free form field extending to the end of the line
# MTAG T1 T2 NT P Comment
# LOC 30 100 0 6 0 Lines of code/function
# LOCm 500 2000 0 6 0 Lines of code/module
# LOCp 999999 999999 0 6 0 Lines of code/project
# MUC 10 33 0 6 0 Cyclomatic complexity/function
# MUCm 200 1000 0 6 0 Cyclomatic complexity/module
# MUCp 999999 999999 0 6 0 Cyclomatic complexity/project
# COM 999999 999999 0 6 0 Comment lines
# M_C 5 10 5 6 3 McCabe/Complexity (visible)
# L_C 7 33 20 6 3 McCabe/Complexity (overall)
# CGS 10 33 0 6 3 Card & Glass Structure: Complexity
# CGSV 7 29 0 6 3 Card & Glass Structure: Complexity (visible)
# CGSc 7 29 0 6 3 Card & Glass Structure: Complexity (concrete)
# FE 12 29 0 6 0 Fan-in (overall)
# FIV 6 12 0 6 0 Fan-in (visible only)
# FIV 6 12 0 6 0 Fan-in (concrete only)
# FQO 12 29 0 6 0 Fan-out (overall)
# FOV 6 12 0 6 0 Fan-out (visible only)
# FOV 6 12 0 6 0 Fan-out (concrete only)
# HRS 100 1000 0 6 0 Henry-Kafura/Shepperd measure (overall)
# HRSV 30 100 0 6 0 Henry-Kafura/Shepperd measure (visible)
# HRSV 30 100 0 6 0 Henry-Kafura/Shepperd measure (concrete)

Ignoring compiler-specific keywords

Some C++ compilers define keywords additional to the ones supported by CCCC, for example the keywords 'far' and 'near' which are used to specify size of pointers in the segmented 16 bit MS/DOS architecture. CCC will normally recognize these keywords as identifiers, and may well be unable to parse some constructs as a result. If problems of this kind occur, it is possible to configure CCCC to ignore the offending keyword, by listing it in the configuration file cccc_ign.dat.

Supporting information

The report generated by CCC contains a section of supporting information which is copied from the file cccc_inf.dat. This file can be changed to present whatever information is required (e.g. make links point to local copy of manual, refer directly to information on metrics).
Getting CCCC

The best place to look for information about CCCC is the CCCC home page at http://www.fse.ac.sydney.edu.au.

Recent versions of CCCC are usually available for download from http://www.fse.ac.sydney.edu.au/pub/littlef.

Significant new production versions are announced in the comp.software.measurement USENET newsgroup.
Appendix D: Analysis of download logs for the project FTP area

The table below was abstracted from raw data supplied by Cole Bergerson, who is webmaster for the host system. The data covers the period November 1997 to November 1998.

<table>
<thead>
<tr>
<th>No. of hits</th>
<th>Item requested</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>/ftp/pub/tlittelf/cccc.lsm</td>
</tr>
<tr>
<td>1</td>
<td>/ftp/pub/tlittelf/cccc_203.tar.gz</td>
</tr>
<tr>
<td>1</td>
<td>/ftp/pub/tlittelf/cccc_203.zip</td>
</tr>
<tr>
<td>1</td>
<td>/ftp/pub/tlittelf/cccc_204.lsm</td>
</tr>
<tr>
<td>154</td>
<td>/pub/tlittelf/LSM-TEMPLATE</td>
</tr>
<tr>
<td>308</td>
<td>/pub/tlittelf/NAMES</td>
</tr>
<tr>
<td>37</td>
<td>/pub/tlittelf/cccc-1.0.tar.gz</td>
</tr>
<tr>
<td>1</td>
<td>/pub/tlittelf/cccc-2.1.1-dos.zip</td>
</tr>
<tr>
<td>4</td>
<td>/pub/tlittelf/cccc-2.1.1-src.tar.gz</td>
</tr>
<tr>
<td>44</td>
<td>/pub/tlittelf/cccc-2.1.1-bin.ELF.tar.gz</td>
</tr>
<tr>
<td>97</td>
<td>/pub/tlittelf/cccc-2.1.1.src.tar.gz</td>
</tr>
<tr>
<td>109</td>
<td>/pub/tlittelf/cccc-2.1.1.zip</td>
</tr>
<tr>
<td>161</td>
<td>/pub/tlittelf/cccc-2.1.2-dos.zip</td>
</tr>
<tr>
<td>36</td>
<td>/pub/tlittelf/cccc-2.1.2-linux.tar.gz</td>
</tr>
<tr>
<td>165</td>
<td>/pub/tlittelf/cccc-2.1.2-src.tar.gz</td>
</tr>
<tr>
<td>54</td>
<td>/pub/tlittelf/cccc-2.1.3-dos.zip</td>
</tr>
<tr>
<td>24</td>
<td>/pub/tlittelf/cccc-2.1.3-linux.tar.gz</td>
</tr>
<tr>
<td>62</td>
<td>/pub/tlittelf/cccc-2.1.3-src.tar.gz</td>
</tr>
<tr>
<td>Path</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc-2.1.4-dow.zip</td>
<td>ccc-2.1.4 dow</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc-2.1.4-linux.tar.gz</td>
<td>ccc-2.1.4 linux</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc-2.1.4-src.tar.gz</td>
<td>ccc-2.1.4 src</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc.lam</td>
<td>ccc.lam</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_191.tar.gz</td>
<td>ccc_191.tar.gz</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_193.tar.gz</td>
<td>ccc_193.tar.gz</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_200.tar.gz</td>
<td>ccc_200.tar.gz</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_200.zip</td>
<td>ccc_200.zip</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_202.tar.gz</td>
<td>ccc_202.tar.gz</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_202.zip</td>
<td>ccc_202.zip</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_203.tar.gz</td>
<td>ccc_203.tar.gz</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_203.zip</td>
<td>ccc_203.zip</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_204.lam</td>
<td>ccc_204.lam</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_204.tar.gz</td>
<td>ccc_204.tar.gz</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_204.zip</td>
<td>ccc_204.zip</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_205.htm</td>
<td>ccc_205.htm</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_205.tar.gz</td>
<td>ccc_205.tar.gz</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_205.zip</td>
<td>ccc_205.zip</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_206.tar.gz</td>
<td>ccc_206.tar.gz</td>
</tr>
<tr>
<td>/pub/tlittlef/cccc_206.zip</td>
<td>ccc_206.zip</td>
</tr>
<tr>
<td>/pub/tlittlef/pccs-1.33.tar.gz</td>
<td>pccs-1.33.tar.gz</td>
</tr>
<tr>
<td>/pub/tlittlef/pccs133.zip</td>
<td>pccs133.zip</td>
</tr>
<tr>
<td>/pub/tlittlef/resume.txt</td>
<td>resume.txt</td>
</tr>
</tbody>
</table>
Appendix E: Questionnaire used for practitioner survey

The survey of practitioners described in chapters 5 and 6 was conducted over the Internet, using an HTML form as the survey instrument and collecting the results electronically. The HTML form used JavaScript to implement drop down list, check box and text data entry controls to allow the respondent to enter his/her response to each question. These controls do not translate well into the current document, so no attempt has been made to represent them, however text has been added in italics to explain the input options presented to the respondent at each point.
Software Metrics Usage Survey

The aim of this survey is not only to obtain your opinions on what is and is not useful in the field of software product metrics, but also to attempt to correlate those opinions with data about your working context. The intention is to investigate the hypothesis that different measures are likely to be useful to people working in different professional and social environments. The form consists of the following sections:

- some factual data about yourself
  - employment sector, size of employer, years experience, what kind of projects you work on, what responsibilities you have
- your opinions about your own preferred ways of working
- your opinions about conditions in your workplace
- your opinions on a number of issues relating to software metrics
  - general questions about use of metrics
  - specific questions on LOC/MVG
  - specific questions on Henry and Kafura's information flow metrics
  - specific questions on Chidamber and Kemerer's metrics suite for OO design
  - specific questions on modified metrics reported by CCCC
- a free format field for any comments you have

Apart from the first and last sections, all questions on the form are in the format of a statement. For each of these statements there is an input element which allows you to indicate your feelings on a five point scale from "strongly agree" through "neutral" to "strongly disagree". If you have difficulty applying any question to your context, please reply "neutral". You may use the free format field at the end to comment on any areas where you wish to amplify your feelings.

The survey is part of the same research project as the development of the freeware CCCC metric analyser tool. Many of the questions relate to metrics which CCCC calculates, but prior use of CCCC (or indeed any automated metric gathering tool) is not a prerequisite for completion of the questionnaire. There are 10 questions on features of CCCC itself to which respondents with no experience of the tool should probably answer 'neutral'.

There is a field in the first section to allow you to enter a return email address. If this is completed one or more reports on the outcomes of the survey may be mailed to you when the data has been gathered and processed. No other use of these email addresses will be made by the research team, and all raw data gathered will be treated in strict confidence. Access to the raw data will be restricted to the researcher, his supervisor and the examiners appointed to assess the project.

The form may appear to be excessive in length; there are 45 statement questions plus the factual ones at the beginning and the free-format field at the end (both of which are completely optional). On the other hand, I don't need people to think too hard about the questions, as the first answer which comes in to your head is more useful than one you have had to think hard about. If you try to spend 10 seconds or less on each statement question, you should find the form comfortably completed in under 15 minutes.

Thanks in advance for your help
Tim Littlefair

---

Important Note

The survey form depends on JavaScript being enabled to work.
If JavaScript is not enabled on your browser, expect some very strange results.
Each question has an associated identifier which is used elsewhere in the current thesis, and is displayed (in italics within square brackets). These identifiers were not visible in the HTML version of the form.

### Factual data about yourself

<table>
<thead>
<tr>
<th>Main source of income (optional) [INCOME]</th>
<th>Private Sector/Government/Education/Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employment status (optional) [EMPSTAT]</td>
<td>Contractor/Employee/Educator/Researcher/Student/Unemployed/Other</td>
</tr>
<tr>
<td>Years of employment in software development (optional)</td>
<td>0 to 1.99/2 to 4.99/5 to 9.99/10 to 14.99/15 or more/prefer not to say</td>
</tr>
<tr>
<td>Your return email address (optional) [EMPSYS]</td>
<td>Free format text field</td>
</tr>
</tbody>
</table>

Each cell in the HTML form version of the next three sections contains a check box control, to allow the respondent to indicate a boolean response to the question.

Which of the following activities are a significant part of your responsibilities (optional)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimate or Tender Preparation [X13]</td>
<td>Project Management (Commercial) [X14]</td>
<td>Project Supervision (Technical) [X15]</td>
<td>Staff Appraisal [X16]</td>
</tr>
</tbody>
</table>

Which of the following languages/techniques/methodologies have you used in the last two years (optional)

|----------|-------|--------|--------|

Which of the following application areas and platforms have you worked in over the last two years (optional)

|-----------------------------|-----------------------------------|----------------------------------|--------------------------------|

For each of the remaining sections up to the free form comment field at the end of the questionnaire, each statement tabulated alongside a drop-down list control which allows the user to select a response from the set 'Disagree Strongly/Disagree/Neutral/Agree/Agree Strongly'. Option 'Neutral' is initially selected.

### Opinions on your preferred ways of working

| I like to choose my own tools [A1] |
| I prefer working in a team to working alone [A2] |
| I would like to spend more time working on documentation [A3] |
| I don't mind depending on other people [A4] |
| I like learning new skills better than exercising old ones [A5] |

235
Opinions on your professional and social environment

My workplace encourages developers to choose their own tools. [H1]
My workplace values talent higher than conformity. [H2]
My workplace gives me sufficient freedom. [H3]
My workplace gives some of my colleagues too much freedom. [H4]
My workplace applies (technical) review techniques to good effect. [H5]

Opinions on metrics and metrics issues

General Issues

| Use software code metrics to evaluate software design. [C1] |
| Use software code metrics to evaluate implementations. [C2] |
| Use software code metrics on my own code. [C3] |
| Use software code metrics on code written by my current peers. [C4] |
| Use software code metrics on code written by my current subordinates. [C5] |
| Use software code metrics on code written by someone else which I now maintain. [C6] |
| Use software code metrics to identify areas for review. [C7] |
| Use software code metrics as an input to personnel appraisal processes. [C8] |
| Use software code metrics to help estimate effort on future projects. [C9] |
| I trust my management not to make inappropriate use of software code metrics. [C10] |

Questions Relating to Procedural Metrics

The following questions relate to the following metrics:
- Lines of Code (LOC)
- McCabe's Cyclomatic Complexity (MVC)
- Lines of Comment (COM)

I find LOC useful as an intuitive guide to the scale of a piece of work. [D1]
I find LOC per procedure or module useful as a predictor of inadequate decomposition. [D2]
I find MVC useful as a predictor of testing difficulty. [D3]
I find MVC per procedure or module useful as a predictor of likelihood of presence of defects. [D4]
I find ratios between COM and MVC useful as predictors of inadequate commenting. [D5]

Questions Relating to Structural Metrics

The following questions relate to the following metrics:
- Fan-in (FI)
- Fan-out (FO)
- Henry and Kafura's Information Flow metric (IF4)

I have an intuitive grasp of the meaning of FI and FO. [E1]
I find FI a useful measure of the width of re-use of a module. [E2]
I have an intuitive grasp of the meaning of IF4. [E3]
I find IF4 a useful measure of the contribution of a module to structural risk for the overall project. [E4]
I believe that calculation of FI, FO, and IF4 from a module's interface only gives enough information to be useful. [E5]
Questions Relating to Object Oriented Metrics

The following questions relate to the following metrics which are all part of Chidamber and Kemerer's Metrics Suite for Object Oriented Design:

- Weighted Methods per Class (WMC)
- Number of Children (NOC)
- Depth of Inheritance Tree (DIT)
- Coupling between Objects (CBO)
- Response for Class (RFC)

Note that the CCCC program does not calculate RFC.

| [I find WMC] useful as a measure of class complexity [F1] |
| [I find NOC useful as a measure of the extent of re-use of a class [F2] |
| [I find DIT useful as a measure of the difficulty of predicting behaviour of a class [F3] |
| [I find CBO useful as a measure of the coupling of a class [F4] |
| [I find RFC a useful predictor of the complexity of the behaviour of a class [F5] |

Questions Relating to modified metrics and other features of CCCC

The following questions relate to the novel metrics calculated by the CCCC metric analysis tool including:

- FL, FO and IF4 modified to account only for visible relationships (FLv, FOv, IF4v)
- FL, FO and IF4 modified to account only for non-abstract relationships (FLc, FOc, IF4c)
- WMC for visible methods (WMCv)

There are also questions about features of the CCCC tool which are intended to make it easier to understand the results of the metric calculation.

| [I have an intuitive grasp of the meaning of the IF4v measure [G1] |
| [I find IF4v a useful predictor of a module's contribution to interface integrity problems for a project [G2] |
| [I have an intuitive grasp of the meaning of the IF4c measure [G3] |
| [I find IF4c a useful predictor of a module's contribution to buildability problems for a project [G4] |
| [I find WMCv a useful predictor of a module's functional importance within a project [G5] |
| [I find colour highlighting of outlying metric values in the report generated by CCCC useful [G6] |
| [I find the ability to customise the thresholds at which colour highlighting applies useful [G7] |
| [I find the ability to generate a single report covering a project and all of its modules useful [G8] |
| [I find the ability to generate a summary report for a project with separate detailed reports on each module useful [G9] |
| [I find the ability to traverse HTML links from the tables generated by CCCC to the source code relating to the current entries in those tables useful [G10] |

Please add your comments here

At this point the form contained a multi-line free-format text entry field. The field supported scrolling, allowing long comments to be entered. These comments form the basis of Appendix G.
Appendix F: Raw survey data

This appendix tabulates the raw responses of the 24 survey respondents to all questions apart from the email address and the free format comment. The email address information will not be disclosed, the comments are reproduced (lightly edited to remove identifying information) in the Appendix G.

Each respondent's answers appear as a single column in the table which appears over the next pages. Each respondent has a nine-digit identifying number (assigned by the CGI script which processed the questionnaire form data). The following codes are used to represent the options offered to the respondent:

- **INCOME**: P=Private Sector; E=Educator; G=Government; O=Other
- **EMPSTAT**: C=Contractor; E=Employee; T=Educator; R=Researcher; S=Student; U=Unemployed; O=Other.
- **EXPYRS**: 1=0 to 1.99; 2=2 to 4.99; 3=5 to 9.99; 4=10 to 14.99; 5=15 or more; X=prefer not to say.
- Series X, Y, and Z are boolean questions, 1 represents true, 0 represents false.
- Series A, B, C, D, E, F, and G are agreement scale questions, 1=Disagree Strongly; 2=Disagree; 3=Neutral; 4=Agree; 5=Agree Strongly.
Table 27:
survey

Raw data

'"

from

quealionnalre


<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Appendix G: Responses to free-format question on survey

The following sections reproduce all of the free-format comments supplied by questionnaire respondents. They have been edited lightly to remove minor spelling and grammatical errors, and to expunge any identifying details, in keeping with the promises of confidentiality which were made in the text of the questionnaire.

These comments are presented as evidence on practitioners' opinions about code measurement in general and the analyzer program produced as part of this project in particular. Some of the comments are complimentary, while others raise criticisms.

I have yet to try CCCP hence "neutral" on the technical aspects in the final set of questions.

I'm doing PhD research at university, and I'm involved in various small software projects (free software). My current area of research is finding the right distribution boundaries for a given program by static analysis of a program's source code. I am looking at software metrics, which originally come from the field of software quality assurance, as a possible hint for predicting the communications patterns at run-time. However, I don't have any concrete results in this area, yet.
In my various software projects, I find myself using LOC rather frequently, but none of the more complex measures. I'm particularly doubtful about numerical ratios between lines of code and lines of comment. In my experience, some code can do without any comments at all (in fact, they would rather harm the understanding, because they might get out of sync, etc.); whereas other code needs A LOT of comments to be maintainable.

Thanks a lot for providing CCCC to the Internet community; it might prove very useful for me in the future. Good luck for your future work.

Sorry but I have not enough experience in CCCC usage to respond to all of the questions.

The last time I used CCCC on my project, many line were uncounted.

note: my project is using extensive template libraries and code.

I am trying the new version as soon as possible.

My platform is NT 4.0 and development tools visual C++.

We do not yet have enough information on the use of metrics here because we are only now beginning to use them.

We may have more meaningful information for you in a few months.
I am not very useful to the survey. I am now retired and do some
shareware programming, but I did work in the field for 40 years (really,
starting with plugboard programming of the IBM 604). I have produced,
alone and within a programming team, many millions of LOC. So I have
answered some questions for current, some for the past - not very useful
for you.

A small observation. I have produced and used myself, written contracts
for the development and use of metric tools, and been privy to the tool
use in a number of large software houses. Everybody has them, everybody
uses them. No one can prove that they are of any use.

One major study covering millions of LOC on communication/telephone
switches which were developed under very exacting data logging found that
the only metric (out of ~20) that had any significant correlation was LOC
to cost - the bigger the code, the more expensive. Nothing else had any
significance.

Now everyone has his favorite, and his justification story, as did this
company since they spent hundreds of millions of dollars collecting this
data, it drove their process. But they proved that it was worthless for
the purpose they wanted (cost/time/testing/error estimation). But do not
let me discourage you, they are still collecting their data, and still do
not know what to do with it.

Answers are probably slightly odd, since I used to use some metrics, and
found them useful in a non-formal way as a good way of finding problem
areas of code. However in my current employment I have been stuck using VB (Arggh. Hate hate) and haven't seen much in the way of useful tools.

Still hope your survey is useful.

Some metrics are not calculated for JAVA code, in the version of CCCC that I currently have.

CCCC is a wonderful effort and I hope to see it flourish. I however, found several defects in CCCC for java ...e.g.: counting return as a decision point and not counting catches. I would also like to have a MSVC++ buildable version...

In relation to the metrics I would much prefer the Shepperd modifications to Henry-Kafura. See Fenton and Pfleeger's Software Metrics: a rigorous and practical approach. pgs 314-316

I would also like to see LCOM and the Basili-Hutchens measure...

The most important thing I would like is a way to feed the calculated measures to a spreadsheet or database to create summaries and charts.

My current job is as an instructor and programming isn't a large portion of my job tasks. However, we keep looking for good tools to apply to student code and suggest for student use. I teach a short block on metrics.

First, I am sorry to fill out your form so lately.
Also, I'm quite busy right now and couldn't "relearn" enough about OO metrics. I am sorry about that.

I hope your project is going well.

Good job. Although I think soft. metrics are extremely dangerous, especially as a way to measure a programmer's performance, they are necessary for dealing with and manipulating a code product as a whole.

CCCC is clearly a low-overhead tool and, as such, doesn't show as much power as the heavily funded packages available. But, it seems solid and is definitely useful.

This is a very useful tool we have been using on a project. It is an EC funded project which is going to help our Company to improve our estimating ability on object-oriented projects.

What we have done is taken past projects, a C++ and a java project. Run CCCC on the source code and used the results to feed into the Predictive Object Points estimating method. This calculated the number of object points, which we then compare to the actual effort taken to complete the project and come out with an Effort per Object point value. This is then used on future projects where we estimate the functionality in terms of object points and the total effort for the project by multiplying them with the effort per object point.

The only thing that would have been excellent is a distinction of the type of function we need this to feed it into the estimating method. The
types we need are Constructors - methods which instantiate an object, 
Destructors - methods which destroy an object, Modifiers - methods which 
change the state of an object and Iterators - methods which accesses all 
parts of an object in a well defined order. 

If we had this information we would be able to accurately calculate the 
weighted methods per class to feed into the algorithm for COPs. 

Other than that the CCC did not work on java files that are about 64k on 
MSDOS it had to be run on Unix. 

Let me know if you are more interested in what we are doing and I am 
happy to give you more information. 

I was shocked when I found CCC as it produced the exact output I was 
looking for. 

I wish I could compile a 32-bit version under NT using Borland, WRVC++, 
or Cygnus GNU-Win32. 

I'd really like to have buttons above the columns in the different 
tables. Pressing the button would sort the rows by that column - most 
likely in descending order. 

Now this is really dreaming, but how about generating data that would let 
an external program generate a directed graph showing the coupling 
between modules?
Would it be hard to add a Python parser?

Thanks for the tool, and good luck in your research.

CCCC does not even measure the *.cc files that are the components of CCCC.EXE. All of the above statements of the survey have the following caveat: If the code to be measured is sufficiently simple that CCCC.EXE can tokenize and parse the code, the metrics are useful and interesting.

I found only about 15% of our 1.5 Million lines of code at my former employer, measurable by CCCC.EXE. The compiler language definition provided in CCCC.EXE is incomplete. The language definition is technically neither C nor C++, but a rather restricted subset of those languages.

I am not disappointed with CCCC.EXE though. How much can a person complain about a software package he did not pay for? I also realize that the problem of measuring code is no simpler and no harder than compiling the code. And approximately 100K lines of C code were measured. None were measured before.

I also enjoyed the tangential research I embarked on into the wild world of compiler design and language translation. I would never have ventured into this area without the impetus of measuring our code.
Appendix H: Materials from the Code Review Experiment

The following pages contain the cover pages for each of the three groups in the experiment, followed by one of the five samples of code as it appears in the group 2 materials (i.e. prefaced by a metrics report). In the group 0 and 1 materials, the source code for this sample appears without the metrics report.

Please read the instructions below, complete the review exercise and then fax or post this page only to

Dr Thomas O'Neill
School of Computer and Information Science
Edith Cowan University
Bradford Street
MOUNT LAWLEY 6050
WESTERN AUSTRALIA
Fax number: (IDD prefix for your location) + 61 8 9370 6100
OR edit the page and email to tim_littlefair@hotmail.com

SOFTWARE ENGINEERING SIMULATED REVIEW EXPERIMENT -
PhD project of Tim Littlefair, supervised by Thomas O'Neill

You have volunteered to participate in this experiment, which is described on the Internet at the location

Briefly, the experiment consists of a simulated software code review, in which the subject is asked to
evaluate a sample of source code against a short checklist of potential quality problems. The subject is
asked to classify each class in the code sample as being either at risk from each issue, or free of risk.

You have been allocated to group 0, who are asked to complete the simulated review exercise in time
comfortable to them.

Please enter your classifications on the table below, using an X to mark the conjunction of a quality issue
and a class which you judge to be at risk in that regard. At the conclusion of the exercise, please record
the actual time taken to the nearest minute.

PLEASE TAKE AS LONG AS YOU NEED TO PERFORM THE EXERCISE TO THE BEST OF
YOUR ABILITY. You may examine the classes or issues in any order you see fit. Please be prepared to
spend up to one hour on the exercise. If at the end of this time you have still not had time to examine some
issues or some classes, just leave them unmarked.

Please indicate classes at risk of each issue with an 'X'

<table>
<thead>
<tr>
<th>Please indicate classes at risk of each issue with an 'X'</th>
<th>class</th>
<th>class</th>
<th>class</th>
<th>class</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excessive length</td>
<td>LearnScreen</td>
<td>TestScreen</td>
<td>VocabElement</td>
<td>Vocabulary</td>
<td>WelcomeScreen</td>
</tr>
<tr>
<td>Excessive decision complexity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inadequate commenting</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cohesion too low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coupling too high</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Respondent's email address: ________________________________

Time taken to complete exercise: _______ minutes

250
Please read the instructions below, complete the review exercise and then fax or post this page only to

Dr Thomas O'Neill
School of Computer and Information Science
Edith Cowan University
Bradford Street
MOUNT LAWLEY 6050
WESTERN AUSTRALIA
Fax number: (IDD prefix for your location) + 61 8 9370 6100
OR edit the page and email to tim.littlefair@hotmail.com

SOFTWARE ENGINEERING SIMULATED REVIEW EXPERIMENT
PhD project of Tim Littlefair, supervised by Thomas O'Neill

You have volunteered to participate in this experiment, which is described on the Internet at the location

Briefly, the experiment consists of a simulated software code review, in which the subject is asked to evaluate a sample of source code against a short checklist of potential quality problems. The subject is asked to classify each class in the code sample as being either at risk from each issue, or free of risk.

You have been allocated to group 1, who are asked to complete the simulated review exercise under a severe time constraint.

Please enter your classifications on the table below, using an X to mark the conjunction of a quality issue and a class which you judge to be at risk in that regard. At the conclusion of the exercise, please record the actual time taken to the nearest minute.

PLEASE PERFORM THE EXERCISE TO THE BEST OF YOUR ABILITY INSIDE A STRICT TIME LIMIT OF 15 MINUTES. You may examine the classes or issues in any order you see fit. If (as is very likely) the time limit expires before you have had time to examine some issues or some classes, just leave them unmarked.

<table>
<thead>
<tr>
<th>Please indicate classes of risk for each issue with an 'X'</th>
<th>class</th>
<th>class</th>
<th>class</th>
<th>class</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>excessive length</td>
<td>LearnScreen</td>
<td>TestScreen</td>
<td>VocabElement</td>
<td>Vocabulary</td>
<td>WelcomeScreen</td>
</tr>
<tr>
<td>excessive decision complexity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>inadequate commenting</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cohesion too low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>coupling too high</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Respondent's email address: _______________________________________

Time taken to complete exercise: ____________ minutes

251
Please read the instructions below, complete the review exercise and then fax or post this page only to

Dr Thomas O'Neill
School of Computer and Information Science
Edith Cowan University
Bradford Street
MOUNT LAWLEY 6050
WESTERN AUSTRALIA
Fax number: (IDD prefix for your location) + 61 8 9270 6109
OR edit the page and email to tim_littlefair@hotmail.com

SOFTWARE ENGINEERING SIMULATED REVIEW EXPERIMENT -
PhD project of Tim Littlefair, supervised by Thomas O'Neill

You have volunteered to participate in this experiment, which is described on the Internet at the location http://www.science.cowan.edu.au/~littlef/CodeReviewExperiment.htm

Briefly, the experiment consists of a simulated software code review, in which the subject is asked to evaluate a sample of source code against a short checklist of potential quality problems. The subject is asked to classify each class in the code sample as being either at risk from each issue, or free of risk.

You have been allocated to group 2, who are asked to complete the simulated review exercise under a severe time constraint, but with the availability of automatically generated reports on numeric attributes of the code to be reviewed.

Please enter your classifications on the table below, using an X to mark the conjunction of a quality issue and a class which you judge to be at risk in that regard. At the conclusion of the exercise, please record the actual time taken to the nearest minute.

**PLEASE PERFORM THE EXERCISE TO THE BEST OF YOUR ABILITY INSIDE A STRICT TIME LIMIT OF 15 MINUTES.** You may examine the classes or issues in any order you see fit. If (as is very likely) the time limit expires before you have had time to examine some issues or some classes, just leave them unmarked.

<table>
<thead>
<tr>
<th>Please indicate classes at risk of each issue with an X</th>
<th>class LearntScreen</th>
<th>class TextScreen</th>
<th>class VocabElement</th>
<th>class Vocabulary</th>
<th>class WelcomeScreen</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excessive length</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Excessive decision complexity</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inadequate commenting</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cohesion too low</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coupling too high</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Respondent's email address: ________________________________

Time taken to complete exercise: ______________ minutes
Detailed report on module TestScreen

<table>
<thead>
<tr>
<th>Metric</th>
<th>LOC</th>
<th>Tag</th>
<th>Overall</th>
<th>Per Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>LOC</td>
<td></td>
<td>120</td>
<td>******</td>
</tr>
<tr>
<td>McCabe's Cyclomatic Number</td>
<td>MVG</td>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Lines of Comment</td>
<td>COM</td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>LOC/COM</td>
<td>L/C</td>
<td></td>
<td>10.909</td>
<td></td>
</tr>
<tr>
<td>MVG/COM</td>
<td>M/C</td>
<td></td>
<td>0.818</td>
<td></td>
</tr>
<tr>
<td>Weighted Methods per Class (weighting = unity)</td>
<td>WMC1</td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Weighted Methods per Class (weighting = visible)</td>
<td>WMCv</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Depth of Inheritance Tree</td>
<td>DIT</td>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Number of Children</td>
<td>NOC</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Coupling between objects</td>
<td>CBO</td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Information Flow measure (inclusive)</td>
<td>IF4</td>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Information Flow measure (visible)</td>
<td>IF4v</td>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Information Flow measure (concrete)</td>
<td>IF4c</td>
<td></td>
<td>0</td>
<td>******</td>
</tr>
</tbody>
</table>

Definitions and Declarations

<table>
<thead>
<tr>
<th>Description</th>
<th>LOC</th>
<th>MVG</th>
<th>COM</th>
<th>L/C</th>
<th>M/C</th>
</tr>
</thead>
<tbody>
<tr>
<td>definition</td>
<td>TestScreen.java:12</td>
<td>19</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Functions

<table>
<thead>
<tr>
<th>Function prototype</th>
<th>LOC</th>
<th>MVG</th>
<th>COM</th>
<th>L/C</th>
<th>M/C</th>
</tr>
</thead>
<tbody>
<tr>
<td>TestScreen( )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>definition</td>
<td>TestScreen.java:40</td>
<td>36</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>actionPerformed(ActionEvent)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>definition</td>
<td>TestScreen.java:99</td>
<td>56</td>
<td>9</td>
<td>9</td>
<td>6.222</td>
</tr>
<tr>
<td>setScreen( LanguageTutorFrame, LanguageTutorFrame)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>definition</td>
<td>TestScreen.java:167</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>setVocabulary(Vocabulary)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>definition</td>
<td>TestScreen.java:173</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Relationships

<table>
<thead>
<tr>
<th>Clients</th>
<th>Suppliers</th>
</tr>
</thead>
</table>

253
<table>
<thead>
<tr>
<th>Member Variable</th>
<th>Class/Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tutor</td>
<td>Tutor.java 15</td>
<td>Member Variable</td>
</tr>
<tr>
<td>TextField</td>
<td>TextScreen.java 15</td>
<td>Member Variable</td>
</tr>
<tr>
<td>TextArea</td>
<td>TextScreen.java 16</td>
<td>Member Variable</td>
</tr>
<tr>
<td>Button</td>
<td>TestScreen.java 16</td>
<td>Member Variable</td>
</tr>
<tr>
<td>Label</td>
<td>TestScreen.java 17</td>
<td>Member Variable</td>
</tr>
<tr>
<td>LanguageTreeFrame</td>
<td>TestScreen.java 18</td>
<td>Member Variable</td>
</tr>
<tr>
<td>Vocabulary</td>
<td>TestScreen.java 19</td>
<td>Member Variable</td>
</tr>
<tr>
<td>ActionListener</td>
<td>TestScreen.java 20</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 21</td>
<td>Member Variable</td>
</tr>
<tr>
<td>VocabElement</td>
<td>TestScreen.java 22</td>
<td>Member Variable</td>
</tr>
<tr>
<td>WindowListener</td>
<td>TestScreen.java 23</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 24</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 25</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 26</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 27</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 28</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 29</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 30</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 31</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 32</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 33</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 34</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 35</td>
<td>Member Variable</td>
</tr>
<tr>
<td>normal Parameter</td>
<td>TestScreen.java 36</td>
<td>Member Variable</td>
</tr>
</tbody>
</table>
package vocabtutorcase;

import java.awt.*;
import java.awt.event.*;

public class TestScreen extends LanguageTutorFrame implements ActionListener,
WindowListener {

    TextField resultTF = new TextField(16);
    TextArea hintTA = new TextArea("",".10,TextArea.SCROLLBARS_NONE);
    Label resultL = new Label("result");
    Label hintL = new Label("hint");
    Label sourceL = new Label("source");
    Label targetL = new Label("target");

    TextField sourceTF = new TextField(16);
    TextField targetTF = new TextField(16);

    Button quitButton = new Button("Quit");
    Button learnButton = new Button("Learn");
    Button nextButton = new Button("Next");
    Button hintButton = new Button("Hint Please!");
    LanguageTutorFrame learnScreenT;
    LanguageTutorFrame testScreenT;

    boolean firstMessage = true; // allows alternate conciliatory message displays

    public TestScreen() {
        super();
        setSize(600, 420);
        setLayout(null);
        add(sourceL);
        sourceL.setBounds(40, 10, 100, 20);
        add(sourceTF);
        sourceTF.setBounds(40, 10, 100, 20);
        add(targetL);
        targetL.setBounds(40, 160, 100, 20);
        add(targetTF);
        targetTF.setBounds(40, 160, 100, 20);
        add(quitButton);
        quitButton.setBounds(20, 320, 60, 60);
        add(learnButton);
        learnButton.setBounds(120, 320, 60, 60);
        add(nextButton);
        nextButton.setBounds(220, 320, 60, 60);
        add(resultL);
        }
```java
public void actionPerformed(ActionEvent evt)
{
    String next = evt.getActionCommand();
    if (next.equals("Quit"))
    {
        System.exit(0);
    }
    else if (next.equals("Learn"))
    {
        this.hide();
       LearnScreenT.show();
    }
    else if (next.equals("Next"))
    {
        // retrieve a new vocabulary from the Vocabulary
        theElement = theVocab.getRandomElement();
        theElement.newHintSequence();
        // display the Vocabulary
        sourceTF.setText(theElement.getSourceWord());
        // clear all other text fields
        hintTA.setText("");
        targetTF.setText("");  
        resultTF.setText("");
    }
    else if (next.equals("Hint Please!"))
    {
        // check that there is a current element
        if (theElement != null)
        {
            // check that there are hints to give
            if (theElement.moreDisplayHintsStrings())
            {
                hintTA.setText(theElement.getNextDisplayHintString());
            }
            else
            {
                hintTA.setText("No further hints are stored!");
            }
        }
    }
    if (evt.getSource() == targetTF)
    

```
// compare with the source and send a message
if (theElement.getTargetWord().equals(targetTF.getText()))
{
    // any well done
    resultTF.setText("Well done, that is correct!");
}
else
{
    // any try again in one of two ways
    if (firstMessage)
    {
        resultTF.setText("Sorry, please try again");
        firstMessage = false;
    }
    else
    {
        resultTF.setText("Go on try again");
        firstMessage = true;
    }
}
} // end actionPerformed

public void setScreen(LanguageTutorFrame learn, LanguageTutorFrame test)
{
    learnScreenT = learn;
    testScreenT = test;
}

public void setVocabulary(Vocabulary theVocab)
{
    this.theVocab = theVocab;
}
} // end TestScreen
Appendix I: Statistical Worksheets

The statistical worksheets used in the processing of the data from the code review experiment are presented on the following pages.
Table 28: Derivation of Correct Responses

<table>
<thead>
<tr>
<th>CPR(O,q)</th>
<th>N</th>
<th>Count of questions</th>
<th>Count of respondents</th>
<th>Proportion of positive responses over all questions and all respondents</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6.50</td>
</tr>
</tbody>
</table>

This worksheet shows the logic for the derivation of national 'correct' answers from the group 0 responses. The first table shows the count of positive responses for each question from group 0 respondents. We sum these counts and then divide by the potential number of positive responses to obtain a target proportion of questions which we wish to classify positively. This proportion is then converted to an integer number of questions.

The next table shows the effect of classifying questions as positive or negative according to whether the CPR(O,q) is greater than a threshold T, with the T varying from -0.5 (all questions will be classified positive) to 4.5 (all questions will be classified negative). For each value of T, the number of questions classified positive is shown.

Finally, the value of T is selected which gives a count of questions classified positive as close as possible to the target, and the question classifications for this threshold are displayed in the following table.
third table as the derived correct response (DCR) for each question. The values in this table are known as DCR(q), for example DCR(9) is the derived correct response to question 9, which happens to be 0 (i.e., negative).
Table 29: ROC Analysis for Group 0

|   | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| DCR(q) | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 0  | 0  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | 1  | 1  | 0  |
| CPR(0,q) | 2  | 2  | 6  | 0  | 1  | 0  | 3  | 4  | 2  | 0  | 5  | 0  | 2  | 5  | 3  | 0  | 2  | 4  | 0  | 2  | 1  | 1  | 2  | 1  |

This worksheet shows the calculation of the Receiver Operating Characteristics (ROC) statistics for the group 0 respondents.

The ROC statistics are calculated by using the responses of the group to classify questions as positive or negative by applying a varying threshold on the number of responses required to classify a question as positive.

At each level of the threshold, the response to each question is classified as TP, TN, FP, FN, then the false positive fraction and the true positive fraction are calculated using the formulae $\text{FPF} = \frac{FP}{FP+TN}$ and $\text{TPF} = \frac{TP}{TP+FN}$.

Finally, the components of the area under the ROC curve are calculated using the trapezium method. These components are shown in the column headed AUC, and their sum, which is the area under the curve calculated for this method is shown at the bottom of the column.
Table 30: ROC Analysis for Group 1

|     | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|-----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| DCR(q) | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| CPR(l,q) | 0 | 1 | 1 | 2 | 2 | 0 | 1 | 3 | 2 | 2 | 2 | 0 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 3 | 0 | 0 | 0 | 1 | 1 | 2 | 1 |

ROC (Group 1). See comments on the worksheet ROC (Group 0) for explanation of the meaning of this worksheet.
Table 3.1: ROC Analysis for Group 2

|         | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 |
|---------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| UCH(2) | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| CPR(2,3)| 5 | 0 | 4 | 1 | 2 | 0 | 0 | 3 | 1 | 1 | 5 | 2 | 1 | 2 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 2 | 3 | 0 | 0 |

<table>
<thead>
<tr>
<th>Y</th>
<th>TP</th>
<th>TN</th>
<th>FP</th>
<th>FN</th>
<th>TPF</th>
<th>AUC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9</td>
<td>0</td>
<td>16</td>
<td>16</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>2</td>
<td>.50</td>
<td>.78</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>11</td>
<td>5</td>
<td>5</td>
<td>.31</td>
<td>.44</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>13</td>
<td>3</td>
<td>7</td>
<td>.16</td>
<td>.22</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>15</td>
<td>1</td>
<td>6</td>
<td>.05</td>
<td>.11</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>15</td>
<td>1</td>
<td>9</td>
<td>.05</td>
<td>.30</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>16</td>
<td>0</td>
<td>9</td>
<td>.00</td>
<td>.30</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>9</td>
<td>.00</td>
<td>.30</td>
</tr>
</tbody>
</table>

ROC (Group 2). See comments on the worksheet ROC (Group 1) for explanation of the meaning of this worksheet.
<table>
<thead>
<tr>
<th>Group</th>
<th>Rowsum</th>
<th>Colsum</th>
<th>Fe</th>
<th>$\frac{(o-e)^2}{e}$</th>
<th>$\frac{(o-e)^2}{e}$</th>
<th>p-value</th>
<th>Critical Value @ 0.05 level of probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP1</td>
<td>13</td>
<td>100</td>
<td>28</td>
<td>12.4444</td>
<td>0.3056</td>
<td>0.0248</td>
<td>7.8209 (Sprintall, 1993, p 486)</td>
</tr>
<tr>
<td>TN1</td>
<td>54</td>
<td>100</td>
<td>127</td>
<td>56.4444</td>
<td>5.9753</td>
<td>0.1059</td>
<td></td>
</tr>
<tr>
<td>FP1</td>
<td>18</td>
<td>100</td>
<td>35</td>
<td>15.5556</td>
<td>5.3753</td>
<td>0.3411</td>
<td></td>
</tr>
<tr>
<td>FN1</td>
<td>15</td>
<td>100</td>
<td>35</td>
<td>15.5556</td>
<td>5.3056</td>
<td>0.3198</td>
<td></td>
</tr>
<tr>
<td>TP2</td>
<td>15</td>
<td>125</td>
<td>28</td>
<td>15.5556</td>
<td>5.5056</td>
<td>0.3198</td>
<td></td>
</tr>
<tr>
<td>TN2</td>
<td>73</td>
<td>125</td>
<td>127</td>
<td>70.5556</td>
<td>5.9753</td>
<td>0.0647</td>
<td></td>
</tr>
<tr>
<td>FP2</td>
<td>17</td>
<td>125</td>
<td>35</td>
<td>19.4444</td>
<td>5.9753</td>
<td>0.3675</td>
<td></td>
</tr>
<tr>
<td>FN2</td>
<td>20</td>
<td>125</td>
<td>35</td>
<td>19.4444</td>
<td>3.0866</td>
<td>0.0159</td>
<td></td>
</tr>
</tbody>
</table>

**chi-squared = 0.9623**

**degrees of freedom = 3**

**Critical value of chi-squared @ 0.05 level of probability = 7.8209 (Sprintall, 1993, p 486)**

=> we accept the null hypothesis, there is no significant difference between groups 1 and 2
List of References


