The Automated Analysis of Object-Oriented Designs

Colin Kirsopp

A thesis submitted in partial fulfilment of the requirements of the degree of Doctor of Philosophy

September 2001

Bournemouth University

The Automated Analysis of Object-Oriented Designs

Colin Kirsopp

A thesis submitted in partial fulfilment of the requirements of the degree of Doctor of Philosophy

September 2001

Bournemouth University

Abstract

This thesis concerns the use of software measures to assess the quality of object-oriented designs. It examines the ways in which design assessment can be assisted by measurement and the areas in which it can't. Other work in software measurement looks at defining and validating measures, or building prediction systems. This work is distinctive in that it examines the use of measures to help improve design quality during design time.

To evaluate a design based on measurement results requires a means of relating measurement values to particular design problems or quality levels. Design heuristics were used to make this connection between measurement and quality. A survey was carried out to find suggestions for guidelines, rules and heuristics from the OO design literature. This survey resulted in a catalogue of 288 suggestions for OO design heuristics. The catalogue was structured around the OO constructs to which the heuristics relate, and includes information on various heuristic attributes. This scheme is intended to allow suitable heuristics to be quickly located and correctly applied.

Automation requires tool support. A tool was built which augmented the functionality available in existing measurement tools by allowing user definable measures and measure sets, and taking input from multiple sources of design information (e.g., CASE tools and source code).

The work described so far presents a potential method for automated design assessment and provides the means of automation. An empirical study was then required to consider the efficacy of the method and evaluate the novel features of the tool.

A case study was used to explore the approach taken by, and evaluate the effectiveness of, 15 subjects using measures and heuristics to assess the design of a small OO system (15 classes). This study showed that semantic heuristics tended to highlight significant problems, but where attempts were made to automate these it often led to false problems being identified. This result, along with a previous finding that around half of quality criteria are not automatically assessable at design time, strongly suggests that people are still a necessary part of design assessment. The main result of the case study was that the subjects correctly identified 90% of the major design problems and were very positive about their experience of using measurement to support design assessment.

Content Overview

CONTENT OVERVIEW III
TABLE OF CONTENTS V
LIST OF FIGURES XI
LIST OF TABLES XIII
ACKNOWLEDGEMENT XV
AUTHOR'S DECLARATION XVII

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 ASSESSING OBJECT-ORIENTED DESIGNS 9

CHAPTER 3 SOFTWARE MEASUREMENT 53

CHAPTER 4 THE DESIGN MEASUREMENT TOOL 98

CHAPTER 5 THE EMPIRICAL STUDY 127

CHAPTER 6 DISCUSSION 169

APPENDICES 189

APPENDIX A OOD HEURISTICS CATALOGUE 191

APPENDIX B MEASUREMENT THEORY 237

APPENDIX C META-MODELLING OBJECT-ORIENTED DESIGNS 241

APPENDIX D DESIGN DESCRIPTION LANGUAGE (DDL) 257

APPENDIX E DATA-MODEL DESCRIPTION 265

APPENDIX F SOURCE MAPPINGS 273

APPENDIX G QUESTIONNAIRE 281

APPENDIX H DESIGN DOCUMENTS 289

APPENDIX I SOURCE CODE FOR THE 'SECOND LANGUAGE VOCABULARY TUTOR' 297

APPENDIX J CASE STUDY TASK 331

APPENDIX K EXAMPLE HEURISTIC ANALYSIS 335

APPENDIX L DATA MODEL FOR THE CASE STUDY DATABASE 339

APPENDIX M DESCRIPTION OF METRICS USED 339

REFERENCES 343

Blank

In

Original

Table of Contents

CONTENT OVERVIEW III
TABLE OF CONTENTS V
LIST OF FIGURES XI
LIST OF TABLES XIII
ACKNOWLEDGEMENT XV
AUTHOR'S DECLARATION XVII

CHAPTER 1 INTRODUCTION 1

1.1 MOTIVATION FOR THE WORK 1
1.2 STATEMENT OF THE PROBLEM 2
1.2.1 Aims of the work 2
1.2.2 Scope and limitations 2
1.3 POSITION OF THE RESEARCH 3
1.4 METHOD 4
1.5 THESIS STRUCTURE 6

CHAPTER 2 ASSESSING OBJECT-ORIENTED DESIGNS 9

2.1 WHAT IS SOFTWARE QUALITY 9
2.1.1 Validation 9
2.1.2 Verification 10
2.1.3 Evaluation 11
2.2 SOFTWARE QUALITY MODELLING 16
2.2.1 McCalls quality factors 16
2.2.2 COQUAMO 18
2.2.3 Gilb's quality specification 22
2.2.4 Quality factor deployment (QFD) 24
2.2.5 Goal question metric paradigm (GQM) 25
2.3 QUALITY ASSESSMENT FOR OO DESIGN 27
2.3.1 What is quality in design? 27
2.3.2 Object-oriented design 27
2.3.2.1 Elements of a design method 27
2.3.2.2 The OO design process 28
2.3.3 What is quality in object-oriented design? 30
2.3.4 OO design heuristics 31
2.3.4.1 What are design heuristics? 31
2.3.4.2 The heuristics survey 31
2.3.4.3 Where do design heuristics come from? 32
2.3.4.4 Heuristic assessment types 33
2.3.4.5 Relationship between heuristic and metrics 34
2.3.4.6 Relationship between heuristic and patterns 36
2.3.4.7 Contradictions and trade-offs 37
2.3.4.8 Validity 38
2.3.4.9 Application of heuristics 39
2.3.4.10 The heuristics catalogue 41
2.4 AUTOMATION IN OO DESIGN ASSESSMENT 42
2.5 SUMMARY 48

CHAPTER 3 SOFTWARE MEASUREMENT 53

3.1 A BRIEF HISTORY OF OO MEASURES 53
3.2 INTRODUCTION TO MEASUREMENT 56
3.2.1 What is measurement? 56
3.2.2 Why measure? 58
3.2.3 Terminology in software measurement 59
3.2.3.1 Direct / Indirect 59
3.2.3.2 Internal / External 59
3.2.3.3 Objective / Subjective 59

3.2.3.4 Product / Resource / Project 60
3.2.3.5 Design / Code 61
3.2.3.6 Theoretical / Empirical 61
3.2.3.7 Measure or metric? 62
3.2.4 Measurement and modelling 62
3.2.5 Measuring object-oriented systems 67
3.3 MEASUREMENT VALIDATION 68
3.3.1 Axiomatic / Property-based measurement validation 68
3.3.2 Algebraic validation 69
3.3.3 Dimensional analysis 69
3.3.4 Measurement theory 70
3.3.5 On the application of measurement theory to software 71
3.3.5.1 Empirical relations and intuition 71
3.3.5.2 The prescription/proscription of analysis techniques based on scale type 71
3.3.5.3 The requirement for complexity measures to be additive 74
3.3.6 Empirical validation 76
3.3.6.1 Techniques for empirical validation 76
3.3.6.2 Combining empirical studies 77
3.3.6.3 Current state of empirical research 78
3.3.7 Measurement types and appropriate validation techniques 79
3.4 MEASUREMENT PROCESSES 80
3.4.1 The measure development process 80
3.4.1.1 Direct measures 81
3.4.1.2 Indirect measures 88
3.4.2 The measure use process 89
3.4.2.1 Establish measurement goals 90
3.4.2.2 Selecting measures 92
3.4.2.3 Collecting measurement results 93
3.4.2.4 Analysing measurement results 94
3.4.2.5 Applying measurement results 94
3.4.2.6 Assessing goal achievement 94
3.4.2.7 Changing process goals 95
3.4.3 Comparing measurement processes and metrics programmes 95
3.5 SUMMARY 96

CHAPTER 4 THE DESIGN MEASUREMENT TOOL 98

4.1 RATIONALE FOR DEVELOPING THE TOOL 98
4.2 PRELIMINARY TOOL REQUIREMENTS 99
4.3 INVESTIGATIVE PROGRAMMING 101
4.4 DOMAIN ANALYSIS 102
4.5 ARCHITECTURAL DESIGN 102
4.5.1 High-level design 102
4.5.2 Object-oriented design representation 104
4.6 TOOL IMPLEMENTATION 105
4.7 TOOL DESCRIPTION 108
4.8 EVALUATION OF THE TOOL 113
4.8.1 Verification of measurement results 114
4.8.1.1 Possible types of verification 114
4.8.1.2 Comparison against hand calculation 115
4.8.1.3 Comparison against other tools 115
4.8.2 Assessing the functionality of the tool 119
4.8.3 Lessons learned from the tool development 122
4.9 FUTURE DEVELOPMENTS 123
4.10 SUMMARY 124

CHAPTER 5 THE EMPIRICAL STUDY 127

5.1 RELATED EMPIRICAL WORK 127
5.2 DESIGNING THE STUDY 129
5.2.1 Choosing a research technique 129
5.2.2 The study's questions 133
5.2.3 Unit of analysis 133

5.2.4 Data collection techniques *134*
5.2.5 Design of debriefing questionnaires *135*
5.2.5.1 Information to be elicited 135
5.2.5.2 Question design 135
5.2.6 Validity *136*
5.3 CONDUCTING THE STUDY 137
5.3.1 The subjects *137*
5.3.2 Preparation *139*
5.3.3 The task *140*
5.3.4 Data collection *141*
5.4 RESULTS 144
5.4.1 Metric selection *145*
5.4.2 Metric usage (by metric) *147*
5.4.3 Metric usage (by subject) *150*
5.4.4 Issues *151*
5.4.5 Heuristics *155*
5.4.5.1 Analysis of heuristics used 155
5.4.5.2 Stand-alone heuristic use 157
5.4.6 Differences between students and industrialists *159*
5.4.7 Subjects' views on measurement *160*
5.5 DISCUSSION 162
5.5.1 Summary of results *162*
5.5.2 Threats to validity *164*
5.5.3 Future work *165*
5.5.4 Conclusions *166*

CHAPTER 6 DISCUSSION 169

6.1 SUMMARY OF WORK DONE 169
6.2 SUMMARY OF FINDINGS 171
6.2.1 Assessing OO designs *171*
6.2.2 Software measurement *173*
6.2.3 Measurement tool *174*
6.2.4 Case study *176*
6.3 EVALUATION OF THE RESEARCH WORK 179
6.3.1 Assessing object-oriented design *179*
6.3.2 Software measurement *180*
6.3.3 The design measurement tool *180*
6.3.4 The empirical study *182*
6.4 FURTHER WORK 185
6.5 CONCLUSIONS 187
APPENDICES 189

APPENDIX A OOD HEURISTICS CATALOGUE 191

A.1 HEURISTIC CLASSIFICATION 191
A.2 INHERITANCE 193
A.2.1 Depth of hierarchy *193*
A.2.2 When to use inheritance *195*
A.2.3 What inheritance should model *198*
A.2.4 Multiple inheritance *199*
A.2.5 Miscellaneous inheritance *201*
A.3 AGGREGATION 203
A.4 ASSOCIATION 205
A.5 CLUSTER 205
A.6 CLASS 206
A.7 ABSTRACT CLASSES 209
A.8 COUPLING 210
A.9 COHESION 212
A.10 INFORMATION HIDING 213

A.11 MESSAGE PASSING 215
A.12 SERVICE 216
A.13 ATTRIBUTE 217
A.14 PROTOCOL 218
A.15 POLYMORPHISM 219
A.16 DYNAMIC MODELING 219
A.17 MISCELLANEOUS 220
A.18 SUMMARISED HEURISTICS 221
A.18.1 Introduction *221*
A.18.2 Inheritance *221*
A.18.2.1 Depth of inheritance 221
A.18.2.2 When to use inheritance 222
A.18.2.3 What inheritance should model 223
A.18.2.4 Multiple inheritance 224
A.18.2.5 Miscellaneous inheritance 225
A.18.3 Aggregation *225*
A.18.4 Associations *226*
A.18.5 Clusters *227*
A.18.6 Classes *227*
A.18.7 Abstract classes *229*
A.18.8 Coupling *229*
A.18.9 Cohesion *230*
A.18.10 Information hiding *231*
A.18.11 Message passing *231*
A.18.12 Services *232*
A.18.13 Attributes *233*
A.18.14 Protocol *233*
A.18.15 Polymorphism *233*
A.18.16 Dynamic modelling *234*
A.18.17 Miscellaneous *234*

APPENDIX B MEASUREMENT THEORY 237

B.1 REPRESENTATIONAL THEORY 237
B.1.1 Relational systems *237*
B.1.2 Empirical relational system *237*
B.1.3 Formal relational system *238*
B.1.4 Representational Theorem *238*
B.1.5 Uniqueness Condition *238*
B.1.6 Scales *238*
B.2 SCALE TYPES 239

APPENDIX C META-MODELLING OBJECT-ORIENTED DESIGNS 241

C.1 INPUTS TO THE MODEL 241
C.2 THE UML META-MODEL 242
C.3 METRICS CONSIDERATIONS 244
C.4 DATABASE CONSIDERATIONS 247
C.5 ANALYSIS TOOL CONSIDERATIONS 247
C.6 MODELLING CONSIDERATIONS 248
C.6.1 Paring the UML meta-model *248*
C.6.2 Converting UML to an RDBMS schema *248*
C.6.2.1 Classes 249
C.6.2.2 Template classes 249
C.6.2.3 Attributes 249
C.6.2.4 Operations 250
C.6.2.5 Associations 250
C.6.2.6 Composition 251
C.6.2.7 Link classes 251
C.6.2.8 Inheritance 251
C.6.2.9 Standard Elements 252
C.7 MAPPING THE COMPOSITE META-MODEL ONTO THE RELATIONAL MODEL 253

C.7.1 Simplifying the UML structural packages *253*
C.7.2 Mapping the simplified UML structural packages into ERDs *254*
C.7.2.1 Mapping associations and aggregations 254
C.7.2.2 Collapsing the inheritance hierarchy 255

APPENDIX D DESIGN DESCRIPTION LANGUAGE (DDL) **257**

APPENDIX E DATA-MODEL DESCRIPTION **265**

E.1 TABLE DEFINITIONS 265
E.2 PERMISSIBLE VALUES 269

APPENDIX F SOURCE MAPPINGS **273**

F.1 JAVA SOURCE-CODE MAPPING 273
F.1.1 Packages *273*
F.1.2 Compilation units *273*
F.1.3 Classes *274*
F.1.4 Interfaces *275*
F.1.5 Extends *275*
F.1.6 Implements *276*
F.1.7 Fields *277*
F.1.8 Methods *277*
F.1.9 Constructors *278*
F.1.10 Parameters *278*
F.1.11 Code Blocks *279*
F.2 JAVA BYTE-CODE MAPPING 280
F.3 RATIONAL ROSE MAPPING 280

APPENDIX G QUESTIONNAIRE **281**

APPENDIX H DESIGN DOCUMENTS **289**

H.1 CLASS DIAGRAM 293
H.2 BEHAVIOURAL DIAGRAMS 294

APPENDIX I SOURCE CODE FOR THE 'SECOND LANGUAGE VOCABULARY TUTOR' **297**

APPENDIX J CASE STUDY TASK **331**

APPENDIX K EXAMPLE HEURISTIC ANALYSIS **335**

K.1 MEASUREMENT RESULTS 335
K.2 REAL PROBLEMS FOUND 336
K.3 FALSE PROBLEMS FOUND 338

APPENDIX L DATA MODEL FOR THE CASE STUDY DATABASE **339**

APPENDIX M DESCRIPTION OF METRICS USED **339**

REFERENCES 343

Blank

In

Original

List of Figures

FIGURE 2-1 MCCALL'S SOFTWARE QUALITY FACTORS 16
FIGURE 2-2 HOW MCCALL'S QUALITY CRITERIA INFLUENCE HIS QUALITY FACTORS 17
FIGURE 2-3 KITCHENHAM'S MODEL OF QUALITY FACTORS, CRITERIA, METRICS AND THEIR RELATIONSHIP (KITCHENHAM 1987) 20
FIGURE 2-4 A QUALITY DEFINITION FRAMEWORK (SHEPPERD 1995) 22
FIGURE 2-5 QUALITY FACTOR TRADE-OFF MATRIX (SHEPPERD 1995) 23
FIGURE 2-6 AN EXAMPLE OF A QFD MATRIX (SHEPPERD 1995) 24
FIGURE 2-7 A GQM HIERARCHY 26
FIGURE 2-8 MANAGEMENT AND DESIGN PROCESS (COOK AND DANIELS 1994) 29
FIGURE 3-1 THE RELATIONSHIP BETWEEN A MODEL AND 'REALITY' (SHEPPERD AND INCE 1993) 63
FIGURE 3-2 EXTENDED VIEW OF MODEL RELATIONSHIPS 64
FIGURE 3-3 MEASURE DEVELOPMENT PROCESS FOR DIRECT MEASURES 81
FIGURE 3-4 MEASURE DEVELOPMENT PROCESS FOR INDIRECT MEASURES 88
FIGURE 3-5 MEASURE USE PROCESS 90
FIGURE 4-1 SCHEMATIC OF DESIGN DATA INPUT 102
FIGURE 4-2 MEASUREMENT TOOL ARCHITECTURE 104
FIGURE 4-3 SCREEN SHOT OF METRIC DEFINITION 109
FIGURE 4-4 SCREEN SHOT OF METRIC-SET DEFINITION 110
FIGURE 4-5 SCREEN SHOT OF PROJECT SET-UP 111
FIGURE 4-6 SCREEN SHOT OF RESULTS WINDOW 112
FIGURE 4-7 SCREEN SHOT OF HELP SYSTEM 113
FIGURE 5-1 SUBJECT'S PRIOR EXPERIENCE IN OBJECT-ORIENTATION 138
FIGURE 5-2 SUBJECT'S PRIOR EXPERIENCE OF NON-OO DEVELOPMENT 138
FIGURE 5-3 STUDENT'S PRIOR EXPERIENCE OF JAVA AND UML 139
FIGURE 5-4 AN ASPECT OF THE CASE STUDY DATABASE MODEL 143
FIGURE 5-5 RESULTS FROM THE METRIC SELECTION SECTION OF THE QUESTIONNAIRE 145
FIGURE 5-6 METRIC-SETS USED BY SUBJECTS TO SUPPORT THEIR ANALYSIS 147
FIGURE 5-7 METRIC USAGE, BY METRIC AND USAGE TYPE 148
FIGURE 5-8 NUMBER OF METRIC USES AND METRICS USED BY SUBJECT 151
FIGURE 5-9 REAL-ISSUES AND NON-ISSUES BY ENTITY TYPE 154
FIGURE 5-10 ISSUES FOUND AND NOT FOUND BY ENTITY TYPE 155
FIGURE 5-11 HISTOGRAM OF NUMBER OF ISSUES FOUND PER SUBJECT 156
FIGURE 5-12 SUBJECT OVERLAP ON ISSUES 156
FIGURE 5-13 HEURISTICS AND HEURISTIC-USES SHOWN BY ENTITY TYPE 157

FIGURE 5-14 USE-TYPE IN HEURISTICS AND HEURISTIC-USES 158
FIGURE 5-15 SUBJECTS' VIEWS ON MEASUREMENT 161
FIGURE C-1 UML TOP-LEVEL PACKAGES 242
FIGURE C-2 UML FOUNDATIONS PACKAGE 243
FIGURE C-3 UML BEHAVIOURAL PACKAGE 243
FIGURE L-1 DATA MODEL FOR CASE STUDY DATABASE 339

List of Tables

TABLE 3-1 VALIDATION TECHNIQUES FOR DIFFERENT CLASSES OF MEASUREMENT 79
TABLE 3-2 GOAL DEFINITION FRAMEWORK 92
TABLE 5-1 SUMMARY OF TIME SPEND ON THE TASK (IN HOURS) 141
TABLE 5-2 SUBJECTS' OPINIONS ON THE METRICS THEY SELECTED 150
TABLE 5-3 DISTINCT ISSUES BY ISSUE QUADRANT 151
TABLE 5-4 IMPORTANCE LEVEL FOR ISSUES FOUND AND NOT FOUND 152
TABLE 5-5 HEURISTIC-TYPE BY ISSUE QUADRANT 153
TABLE 5-6 HEURISTIC-TYPE BY ISSUE SIGNIFICANCE 153
TABLE 5-7 SUBJECT PERFORMANCE BY HEURISTIC USAGE-TYPE 159
TABLE 5-8 SUMMARY OF ISSUES FOUND BY SUBJECT TYPE 160
TABLE 5-9 SIGNIFICANCE OF ISSUES FOUND BY SUBJECT TYPE 160
TABLE A-1 DEPTH OF INHERITANCE HIERARCHY HEURISTICS 194
TABLE A-2 WHEN TO USE INHERITANCE 196
TABLE A-3 WHAT INHERITANCE SHOULD MODEL 198
TABLE A-4 MULTIPLE INHERITANCE HEURISTICS 199
TABLE A-5 MISCELLANEOUS INHERITANCE HEURISTICS 202
TABLE A-6 AGGREGATION HEURISTICS 204
TABLE A-7 ASSOCIATION HEURISTICS 205
TABLE A-8 CLUSTER HEURISTICS 206
TABLE A-9 CLASS HEURISTICS 208
TABLE A-10 ABSTRACT CLASS HEURISTICS 209
TABLE A-11 COUPLING HEURISTICS 211
TABLE A-12 COHESION HEURISTICS 213
TABLE A-13 INFORMATION HIDING HEURISTICS 214
TABLE A-14 MESSAGE PASSING HEURISTICS 216
TABLE A-15 SERVICE HEURISTICS 217
TABLE A-16 ATTRIBUTE HEURISTICS 218
TABLE A-17 PROTOCOL HEURISTICS 218
TABLE A-18 POLYMORPHISM HEURISTICS 219
TABLE A-19 DYNAMIC MODELLING HEURISTICS 219
TABLE A-20 MISCELLANEOUS HEURISTICS 220
TABLE A-21 DEPTH OF INHERITANCE (SUMMARY) 222
TABLE A-22 WHEN TO USE INHERITANCE (SUMMARY) 223
TABLE A-23 WHAT INHERITANCE SHOULD MODEL (SUMMARY) 223
TABLE A-24 MULTIPLE INHERITANCE HEURISTICS (SUMMARY) 224
TABLE A-25 MISCELLANEOUS INHERITANCE HEURISTICS (SUMMARY) 225

TABLE A-26 AGGREGATION HEURISTICS (SUMMARY) 226
TABLE A-27 ASSOCIATION HEURISTICS (SUMMARY) 226
TABLE A-28 CLUSTER HEURISTICS (SUMMARY) 227
TABLE A-29 CLASS HEURISTICS (SUMMARY) 229
TABLE A-30 ABSTRACT CLASS HEURISTICS (SUMMARY) 229
TABLE A-31 COUPLING HEURISTICS (SUMMARY) 230
TABLE A-32 COHESION HEURISTICS (SUMMARY) 231
TABLE A-33 INFORMATION HIDING HEURISTICS (SUMMARY) 231
TABLE A-34 MESSAGE PASSING HEURISTICS (SUMMARY) 232
TABLE A-35 SERVICE HEURISTICS (SUMMARY) 232
TABLE A-36 ATTRIBUTE HEURISTICS (SUMMARY) 233
TABLE A-37 PROTOCOL HEURISTICS (SUMMARY) 233
TABLE A-38 POLYMORPHISM HEURISTICS (SUMMARY) 234
TABLE A-39 DYNAMIC MODELLING HEURISTICS (SUMMARY) 234
TABLE A-40 MISCELLANEOUS HEURISTICS (SUMMARY) 235
TABLE B-1 SCALE TYPES AND ADMISSIBLE TRANFORMATIONS 239
TABLE B-2 SCALE TYPE AND APPROPRIATE STATISTICS 239
TABLE C-1 META-MODEL ELEMENT USAGE BY MEASURES 246
TABLE K-1 CHIDAMBER AND KEMERER METRICS RESULTS 335
TABLE K-2 SELECTED LORENZ & KIDD CLASS-LEVEL METRICS RESULTS 336
TABLE K-3 LORENZ & KIDD METHOD SIZE METRICS RESULTS (TOP-END SUBSET) 336

Acknowledgement

Firstly, I would like to thank my supervision team Martin Shepperd and Steve Webster for many lively discussions, but mostly for their support and forbearance.

Thanks are also due to those students from the 1997-1998 cohort of the Software Engineering Management degree who attended the Object-Oriented Software Engineering (OOSE) unit. Their earnest efforts made the case study section of this thesis possible.

I would like to thank the School of Design Engineering and Computing for its continued financial support. The OOPS specialist group of the British Computer Society also contributed financial support through their award of the 1996 scholarship for advancement of object technology in the UK.

Blank

In

Original

Author's Declaration

The following publications are based on the work presented in this thesis:

[1] Kirsopp, C., M. Shepperd, and S. Webster. 'An empirical study into the use of measurement to support OO design evaluation', in *Proc. Sixth international software metrics symposium.* Boca Raton, Florida: IEEE Computer Society, 1999.

[2] Kirsopp, C. 'Measurement and the software development process', in *Proc. 12th European Software Control and Metrics Conference.* London, UK: 2001.

[3] Kirsopp, C. and M. Shepperd. 'Using heuristics to assess object-oriented design quality', in *Proc. 5th International Conference on Empirical Assessment & Evaluation in Software Engineering.* Keele University, Staffordshire, UK: 2001.

[4] Kirsopp, C., 'A survey and review of object-oriented design heuristics'. Technical report, ESERG TR-01-04, Bournemouth University, 2001.

Chapter 1 Introduction

1.1 Motivation for the work

Many people have provided convincing arguments of why measurement is important, e.g., (Fenton and Pfleeger 1996; Shepperd 1995). Measurement offers control and sharpens analysis in an objective and repeatable manner. To be able to use measurement with confidence the validation of measures is not sufficient. Users of measurement need to know how they should use measures to achieve their particular goals. To help measurement users with this task, workers in software measurement need to know how people can, and do, use measurements in practice. This will help to identify the information and support required for the successful application of software measures. This thesis seeks to investigate how people use metrics within a particular measurement application domain - object-oriented design assessment.

This work is essentially an investigation into the use of object-oriented design measures for design assessment. There is a great deal of work suggesting new measures or discussing techniques for validation, but less on the practical use of measurement. The existing work on measurement utility focuses mainly upon prediction systems (Zhao, Wohlin et al. 1998; Li and Henry 1993b; Basili, Briand et al. 1996). There is little published work investigating other types of measurement use. This work is intended to start redressing the balance and to encourage other researchers to follow suit.

Many existing measures are described as being an indicator of some or other 'ility'. This is an indication that product assessment might be a fruitful measurement use to investigate. The intention is to investigate the extent to which measurement can be used in assessment and where its limitations lie.

Object-oriented design products were chosen as the focus of the assessment for several reasons. There are a large number of object-oriented design measures in the literature. Assessing earlier lifecycle products allows for cheaper and easier corrective action based on the assessment. This suggests that design products, rather than code, could be most usefully assessed. Object-orientation is also the current vogue for designing systems with UML becoming heavily used in industry.

1.2 Statement of the problem

1.2.1 Aims of the work

The goal is to examine the theoretical limitations and practical considerations of people using measurement to assess design quality in practical situations. This general goal may be broken-down into four more specific aims.

1. To determine the degree to which quality can be assessed at design time and the degree to which this assessment can be automated.
2. To consider mechanisms for achieving design assessment and its automation.
3. To explore the necessary and desirable functionality of tools to support design assessment.
4. To examine empirically how people use measures for design assessment in a practical situation.

1.2.2 Scope and limitations

Some authors have published work attempting to show correlations between particular measures and external quality indicators. They attempt to predict future values for measures such as defect density. That is not the aim of this work. This thesis is primarily concerned with the assessment of the quality of the designs themselves, rather than the prediction of future code quality. The purpose of this assessment is to help improve the quality of OO designs at design time.

The investigation of measurement utility looks at the extent to which measurement can be used to assess quality and the degree to which this assessment can be automated. Essentially, it is an examination of the scope and limitations of software

measurement within the quality assessment of software products. As an illustration of product assessment, the thesis looks at the assessment of design products for object-oriented systems. Although a very particular application of measurement is being examined, and there will certainly be some results specific to object-oriented design, it is hoped that much of the work on how measurement should be utilised will generalise beyond the particular illustrative example.

When the term quality is used within the thesis it has a specific meaning. The thesis deals with what will be termed the 'evaluation' of products, as distinct from validation and verification (V&V). Evaluation is not concerned with ensuring that the design meets the user's real requirements or checking that the design produced matches the specification. It is the assessment of how good the design is relative to the multitude of other possible designs for the same problem (assuming V&V have already been completed). This distinction is discussed more fully in chapter 2.

In examining measurement utility, many of the other aspects often associated with measurement research are also discussed, e.g., metric definition and measurement validation. However, this thesis will not suggest any new measures or perform any measurement validation.

Although it is an aim to empirically examine how people use measures for design assessment in a practical situation, several potentially useful approaches to such an empirical study could be taken. This work will not seek to build cognitive models (Nehaniv 1999) of measurement users or investigate measurement use in the very fine detail provided by methods such as think aloud protocols (Rosson and Gold 1989; von Mayrhauser and Vans 1995). Detailed work of this type may well prove to be useful research, but given the current lack of work in this area it was decided to concentrate on higher-level questions of the subjects' efficacy and general approach.

1.3 Position of the research

The most straightforward way to position the work done for this thesis is within Whitmire's software measurement framework (Whitmire 1997). This framework

identifies three views onto four objects of measurement. The strategic view considers the long-term view of the organisation, the tactical view considers the current project, and the technical view considers specialist design issues. The objects to measure are: processes, the things we do; projects, instances of processes; products, the deliverables created; and resources, such as people or time. The framework has a further dimension that considers the role of measurement. Whitmire identifies the following roles for measurement: estimation, prediction, assessment, comparison and investigation. Assessment is the evaluation of a single software entity without reference to measurement values from any other entity. Comparison is also used for evaluation, but compares the measured values of two or more similar entities. Whitmire makes a distinction between estimation and prediction mainly on the basis that estimation is used to determine likely resource levels and that prediction is used to determine likely future values of product measures. Investigation is the use of measurement data in research to support or refute hypotheses.

This work concerns the measurement of the *products* of object-oriented design. The measurement of processes, projects and resources will not be discussed. Although a limited amount of discussion will be made at the tactical and strategic levels, this work will focus on the *technical view*. In examining the possible applications of object-oriented design measures consideration will be given to all of the roles of measurement suggested by Whitmire, but with particular emphasis on *assessment* and *comparison*.

To recap, this work takes a *technical view* of the measurement of the *products* of object-oriented design to perform *assessment* or *comparison* on those products.

1.4 Method

The aims outlined in section 1.2.1 are not independent. There is a clear flow of work through the list from beginning to end, with later aims depending on the earlier ones. The method of addressing these aims will now be considered aim by aim.

Aim 1: To determine the degree to which quality can be assessed at design time and the degree to which this assessment can be automated.

The first aim is addressed by using the standard McCall quality factors (McCall, Richards et al. 1977) to illustrate the degree to which they can be assessed with the information available in a design model and the extent to which this assessment is objective (and hence, in theory, automatable).

Aim 2: To consider mechanisms for achieving design assessment and its automation.

The second aim is tackled in two ways. Firstly, by looking at existing quality modelling techniques and design heuristics as mechanisms for design assessment. Secondly, by looking at the software measurement literature to assess measurement as a mechanism for automation.

Aim 3: To explore the necessary and desirable functionality of tools to support design assessment.

Automation implies tool support. The third aim is approached by looking at existing suggestions regarding measurement automation and the facilities available in existing measurement tools. Some suggested tool facilities not available in existing tools are shown to be feasible through the production of a new measurement tool.

Aim 4: To examine empirically how people use measures for design assessment in a practical situation.

The fourth aim involves studying the actual use of measures in the design assessment activity. It requires an investigation into how people would actually employ measurement to aid them in design assessment and whether measurement really aided their assessments. This clearly cannot be done theoretically and requires an empirical study. There is a more detailed discussion of the methodology of this study as part of chapter 5.

1.5 Thesis structure

Chapter 1 - Introduction

The introduction is intended to help the reader by providing a context within which to read the main body of the thesis. It positions the work within the wider context, makes a clear statement of the problem being solved. The method that will be followed to tackle the problem is discussed and the structure of the thesis outlined.

Chapter 2 - OO design assessment

This chapter reviews the literature associated with defining and assessing the quality of software products in general and object-oriented design products in particular. The first stated aim of the research is 'to determine the degree to which quality can be assessed at design time and the degree to which this assessment can be automated.' In order to investigate the assessability and automatability of design quality, it is necessary to first consider what is meant by quality. Without a clear picture of the attribute under study the analysis is likely to be meaningless.

The investigation of software quality begins with a consideration of the traditional approach involving validation, verification and evaluation. Previously published methods of modelling or quantifying quality are evaluated. The discussion is then focused more closely on the meaning of quality, and its assessment, when applied to object-oriented designs. This includes forays into the process of object-oriented design and a review of object-oriented design heuristics. The final section of this chapter considers the extent to which quality can be assessed from the information available at design-time and the degree to which this assessment can be automated.

The particular view taken of automation in design assessment is a measurement-centric view. To consider automation of design assessment it is necessary to consider not only the extent to which automation is theoretically possible but also the mechanisms by which automation may be achieved. Consequently, the mechanisms of software measurement also have to be considered.

Chapter 3 - Software measurement

A brief history of OO software measures is presented first. This provides background to the remainder of this chapter and context for the measures used in the case study in chapter 5. An introduction to software measurement is then given. This includes a discussion of what software measurement is and why measurements are taken, as well as an analysis of software measurement terminology. The introduction also discusses the important relationship between measurement and modelling and examines some OO specific issues in measurement.

If measures are being used to assess quality, it is important that they correctly represent the attributes they purport to measure and are useful in the task to which they are being applied. This is the job of measurement validation. The third section of chapter 3 reviews the literature on measurement validation, critically reviewing both theoretical and empirical techniques. A discussion of the processes by which software measures are developed and used is presented in the fourth section.

Chapter 4 - The OO design tool

One point emerging from the discussion of software measurement is the need for automation. After studying existing software measurement tools it becomes apparent that they lack some features helpful for both industrial and academic users, e.g., user definable metrics and cross-source measurement collection. A tool was proposed and built to provide proof of concept that some of these features could be included in a measurement tool and that they would be useful to end-users. This tool is the subject of chapter 4. A description is given of the main development activities including the conception, design, building and testing of the tool. The completed tool is evaluated against the criteria for which it was built and compared with a range of other measurement tools.

Chapter 5 - The empirical study

One of the main aims of the PhD programme is 'to examine empirically how people use measures for design assessment in a practical situation.' Chapter 5 describes this empirical study, which used the tool described in chapter 4. In the study, 15 subjects

(full-time students and industrialists) are asked to assess the architecture of a small system (15 Java classes) using design heuristics and measurement results.

The first section of chapter 5 presents a review of related empirical work on software measurement. This review shows the distinct aims of this study and so highlights the need for a new study rather than the replication of a previous one. The second section deals with the choice of the research technique (a case study was chosen) and gives a description of the design of the study. The design looks at the study's questions, its unit of analysis, and the data collection techniques. The actual conduct of the study is considered in the third section. This deals with a description of the subjects, their preparation for the case study, the task itself and the data collection.

The case study results are based on assessment reports produced by the subjects and a follow-up questionnaire. The case study database has a complex data model populated by a large amount of data. The results section presents analyses of various views into this data. Most of the analysis is focused toward examining the approach the subjects took to the task and how effective they were at using automation to assist in the assessment.

Chapter 6 - Discussion

The discussion begins with a summary of the work done. The findings presented in the thesis are summarised and discussed in the second section. Following this is an evaluation of the work done towards the thesis, which highlights its strengths and weaknesses. Suggestions for further work are then made. These contain possible extensions to the work presented, work to address weaknesses identified by the literature surveys and new work suggested by the work presented in this thesis. A final short section of conclusions completes the thesis. This section sets out what the author believes to be the main achievements of the work.

Chapter 2 Assessing object-oriented designs

The purpose of this chapter is to examine what is meant by quality for software products in general, and object-oriented design products in particular. It is also intended to show the extent to which quality can be assessed from the information contained in a design and the extent to which this assessment can be automated.

The chapter starts with a section discussing the standard elements usually associated with software quality (validation, verification, and evaluation). In the second section, some previous work on attempts to formalise software quality evaluation is examined. The third section deals with aspects of software quality specific to object-oriented design.

2.1 What is software quality

When asking the question 'what is a good object-oriented design?' an attempt is being made to understand how to assess the quality of a design. This section is concerned mainly with the philosophical aspects of the meaning of quality, rather than the detailed process of assessing quality. The starting point for the discussion is a consideration of the traditional approach to software quality involving validation, verification and evaluation.

2.1.1 Validation

Validation is considering whether the system being built is the system the user really wants. The system being built could be the system the producers mistakenly think the user wants, or even the system the user mistakenly thinks he wants. Different individuals in the client body may have different perceptions of their needs from a piece of software. There will also be problems of imprecision in communicating these perceived needs, both within customer groups and to the developers. These types of problem cannot be inferred from the software products themselves and must be handled between the client and developer. An example of validation is the use of

prototyping to aid the discussion of system specifications. Validation is inherently informal and is not automatable.

2.1.2 Verification

Verification is the extent to which the system meets its explicitly stated functional requirements. A system that fully implements the stated requirements would generally be taken to be better than one that only partially covers the requirements. Verification of systems is usually done in an informal manner. Developers examine the software products during reviews and decide whether the system operations provided match-up with the functionality described in the specification.

It should be possible (in principle) to perform a thorough formal verification of a system. Given a formal description of the specification and a formal description of the product, it should be possible to mathematically prove their equivalence. Since this seems such a complete and precise way of checking a system's 'correctness', why aren't all systems produced in this way? Sommerville (2001) states that 'verifying a non-trivial software system takes a great deal of time and requires specialised tools such as theorem provers and mathematical expertise'. Pressman and Ince (2000) list a number of issues such as high start-up costs, avoiding over-formalisation and need for expert training and consultancy that should considered before adopting formal methods. They also state that traditional development methods should not be abandoned, i.e., formal methods are an addition to standard development techniques rather than a replacement for them, and as such, represent an additional cost. The second problem is one of validation. Even if you produce a correct formal description of what you think the user wants, you still can't be sure that it is the system the user really wants. This is particularly true of the formal approach where the domain experts are unlikely to have the expertise to spot errors in a formal specification. Other possible problems include factors such as: requirements creep; and trade-offs between requirements, and cost or delivery time. This means that your precise formal specification may soon no longer describe the system you are trying to build. Formal notations are good at capturing functional requirements but less useful for specifying other types of requirement such as performance or usability. In short, formal

verification is a very time consuming, expensive and difficult task, after which you still can't be sure you have the desired system.

2.1.3 Evaluation

In addition to the above argument for validation, it must be pointed out that there will be many possible solutions, for any given problem, which fulfil the specified requirements. How should the decision be made as to which of these solutions to use? This is the question evaluation attempts to answer.

Evaluation is done by assessing how the system measures up to the implicit requirements of standard quality factors such as maintainability, reusability, understandability etc. Such factors are not always independent of each other and may indeed require trade-off one against the other. In considering the desired qualities for a particular system, the relative importance of the various types of quality need to be considered to help resolve such trade-off questions. This set of 'ilities' and their relative importance levels can be thought of as the system's desired 'quality profile'. The major question in software evaluation is how to assess the various software qualities that comprise this profile.

The creative nature of software design has led to a craft approach to design evaluation, which is generally done in an intuitive way. Those evaluating a solution know it is good because it 'feels right'. Riel (1994) suggests that what they are actually doing is using a set of subconscious heuristics which have been learned through experience (a more formalised approach to heuristic based evaluation will be discussed later). The problem with this type of assessment is that the evaluation of a particular alternative will vary from practitioner to practitioner depending upon their experience.

If the quality of software products is to be usefully assessed, a clear definition is needed of what is meant by quality. Ideally, the assessment of quality should be both objective and repeatable. This would allow quantitative, or at least comparative, judgements to be made consistently by different assessors.

It is intuitively attractive to try to produce a single quality value for a product. This would facilitate the answering of questions such as 'if I were to make such-and-such a change, would the product be improved?' However, because of the complex nature of quality it is difficult to calculate that a product A has a level of quality X, or say that product B is of higher quality than product C. To be able to do this requires a quantifiable definition of quality. Budgen (1994) states that 'the ultimate goal of quality must be that of fitness for purpose'. This is an often-quoted definition of quality, but it is not quantifiable. So how can a number be put to such a multifaceted concept as quality?

A software product can be evaluated from several viewpoints, such as academic, producer, or customer. There are any number of abstract properties which may be considered to have a bearing on quality, such as portability, maintainability, reusability, readability and so on. These properties were chosen to help produce good commercial systems. However, from an academic point of view they are largely considered in isolation from any higher level commercial goals. In academia a product can be considered as 'good' in itself without considering how it might benefit the company producing or using the product.

From this academic perspective, the quality of a product can be assessed by individually assessing each of the software engineering principles. This means producing a separate set of values representing the portability, reusability, maintainability and so on, rather than a single value for quality. These individual measures can be derived from metrics applied to the product.

The quality of a system can also be considered from a producer's viewpoint. At the most fundamental commercial level, the best design may be considered the one that does most to help the company achieve its long-term goals. Yourdon (1994) defined a good design as '... one that balances a series of trade-offs to minimise the total cost of the system over its entire productive life span'. It may be argued, however, that Yourdon's view of minimum cost is too limited a view and may not be the highest priority for a company. The definition seeks only to minimise costs but neglects

maximising returns or minimising risk. These considerations are also important to businesses.

A company about to produce a system will want to know not only, how much will this piece of software cost, but also, how great will the return on the investment be? Moreover, what are the chances of the project failing? Will coding a particular product give a greater or lesser return than this alternative product? Another consideration is that returns on the investment may occur outside of the project itself, and a wider company view of the value of a product may have to be taken. Designing or coding for reuse, for instance, incurs costs during the project that have their pay back outside of the project. As well as the consideration of these direct financial benefits, there are indirect benefits to consider. Choosing a project that moves the company into new market areas, introduces them to new customers, or improves company image, may be considered as more important than immediate financial return.

To enable assessment of this kind of commercially based quality, requires the ability to estimate to the likely costs, returns, and risks of a project based on the abstract values calculated for adherence to the academic principles. This would only be possible by creating models in terms of the measurable properties used in assessing the academic principles. Empirical data could be used to statistically derive the relationships and coefficients between these metrics and the likely costs, returns and risks of a project. These relationships and coefficients may well prove to be company specific and so require calibration on a company by company basis.

The other main perspective from which the quality of a piece of software can be viewed is that of the customer. From this perspective, the view taken is largely that of the observable behaviour of the system. To this extent the customer's view of quality is, at a high level, a subset of the producer's view. This must be so because one of the main goals of the producer is to satisfy the customer needs from the software as laid down in the specification.

The customer is also concerned with interactions between themselves and the producer that are affected by the quality of software. An example of this is maintainability. From the point of view of producers, maintainability might be defined by the likely cost of making a change to the system. From the customer's viewpoint, maintainability might be defined as the time taken for a requested change to be made. While the time taken doesn't depend solely on the quality of the product, the structure of the system architecture will effect the ease with which a change might be made and so, indirectly, the time it will take.

Some researchers have suggested (e.g., (Kitchenham 1987)) that many aspects of quality are only measurable during the active life of a product. An alternative view might be that these attribute levels are intrinsic to the product and independent of what happens during the product lifetime. To take the first view means that assessing these attributes whilst the product is under development is attempting to predict what will happen in the future. To take the second means that when the product is being assessed the measurement/estimate is either a direct or an indirect assessment of an attribute that already exists in the product. Which is the better definition to choose will depend on the goal of the measurement. These ideas can be further examined by using maintainability as an example quality factor.

If the definition of maintainability allows it to be measured before release then it is being defined as an intrinsic property of the system. An operationalised version of a definition such as 'the propensity of the system to be easily maintained' could be used. This is an internal measure and not a prediction of any externally observable or verifiable behaviour (although the result may be used for prediction or its exact definition selected on the basis of predictive power for some other external attribute).

Measuring the maintainability of a system by a metric such as 'maintenance effort per change request' would require us to take the view that this attribute could only be measured during the active lifetime of the product. This definition requires a view that maintainability is dependent not only on the system itself but also on its usage environment. If this definition is used then any attempt to assess maintainability

before release is prediction rather than measurement. The definitions relate to different goals. The first definition relates to the quality of the product itself (useful if the maintenance environment is unknown). The second relates to the ability to be able to allocate funds or estimate costs for maintenance.

What, in an ideal world, would it be most desirable to know about the future maintenance of a product? This question might lead to defining maintainability as 'the total maintenance cost over the system lifetime'. However, some systems could be so unmaintainable that they rapidly cease to be supported. This type of system degradation is described in Leyman's 'law' of increasing complexity (Lehman and Belady 1985). It describes the tendency of system structure to degrade over time unless effort is actively applied to reverse this trend. Even though the cost per change would be high for such unmaintainable systems, they could have a lower total maintenance cost due to the shorter supported lifetime.

A better alternative definition for maintainability would therefore be 'the average cost per change request over the system lifetime'. It is, however, impossible to look into the future to see which changes will be made and what their cost will be. An assessable alternative definition would be 'the estimated average cost of predictable changes'. The costs of the various predicted changes could be weighted by their likelihood of occurrence. With this type of pre-release definition of maintainability, metrics such as 'maintenance effort per change request', measured after release, could then be used to give an indication of how well maintenance is being done. Alternatively, it could be used to decide the accuracy of the original maintainability estimate.

Similar arguments and definitions could be derived for other quality factors; maintainability is used here only as an example. This shows that although words like maintainability are used as if everyone shares and agrees a common definition, this is not case. Workers in software quality may talk with a common vocabulary, but they often intend different meanings. However, it questionable whether a single definition would be desirable, the precise definitions will depend upon the viewpoint from

which the assessment is being made; the environment in which it is being made; and the goal of the assessment. Since many valid viewpoints and goals exist, many valid definitions for quality factors will also exist.

2.2 Software quality modelling

In an attempt to move towards a wider understanding and a more precise definition of quality, several researchers have produced so called 'quality models.' Early work on software quality modelling was done by Boehm, Brown et al. (1978) who mapped primitive characteristics (such as device efficiency, or structuredness) to intermediate characteristics (such as reliability, or modifiability). McCall, Richards et al. (1977) presented their own extended set of quality criteria (primitive characteristics) and quality factors (intermediate characteristics) and standardised[†] the terminology. This is the so-called factor criteria metric (FCM) method.

2.2.1 McCalls quality factors

McCall organised his quality factors into three main areas as shown below.

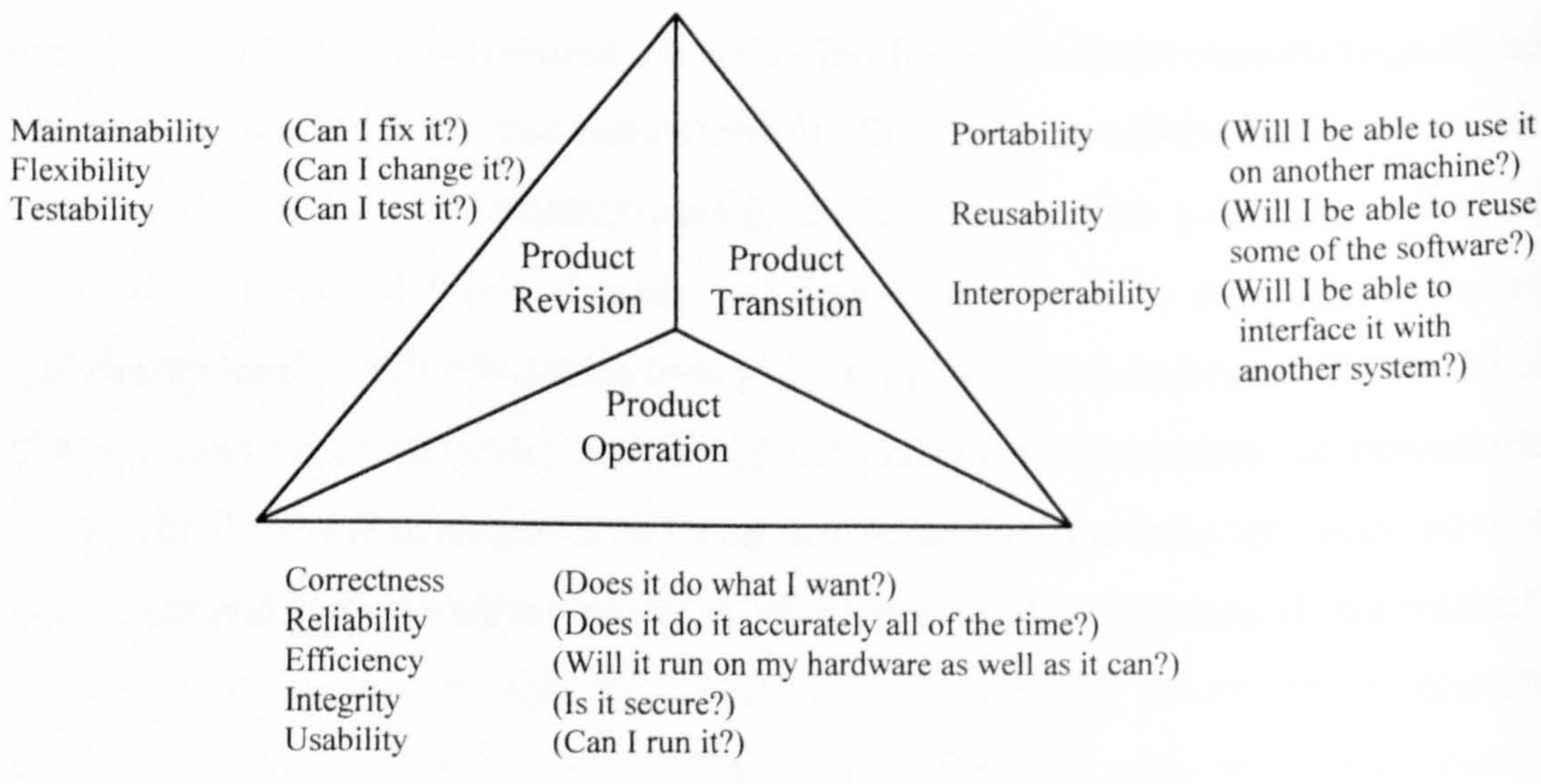


Figure 2-1 McCall's software quality factors

McCall also showed the influences between this set of quality factors and a larger set of quality criteria. These influences are shown in figure 2-2. For a discussion of the meaning of these criteria see section 2.4.

[†] McCall's work forms part of the IEEE Quality Metrics Standard Committee report IEEE P-1061-D2

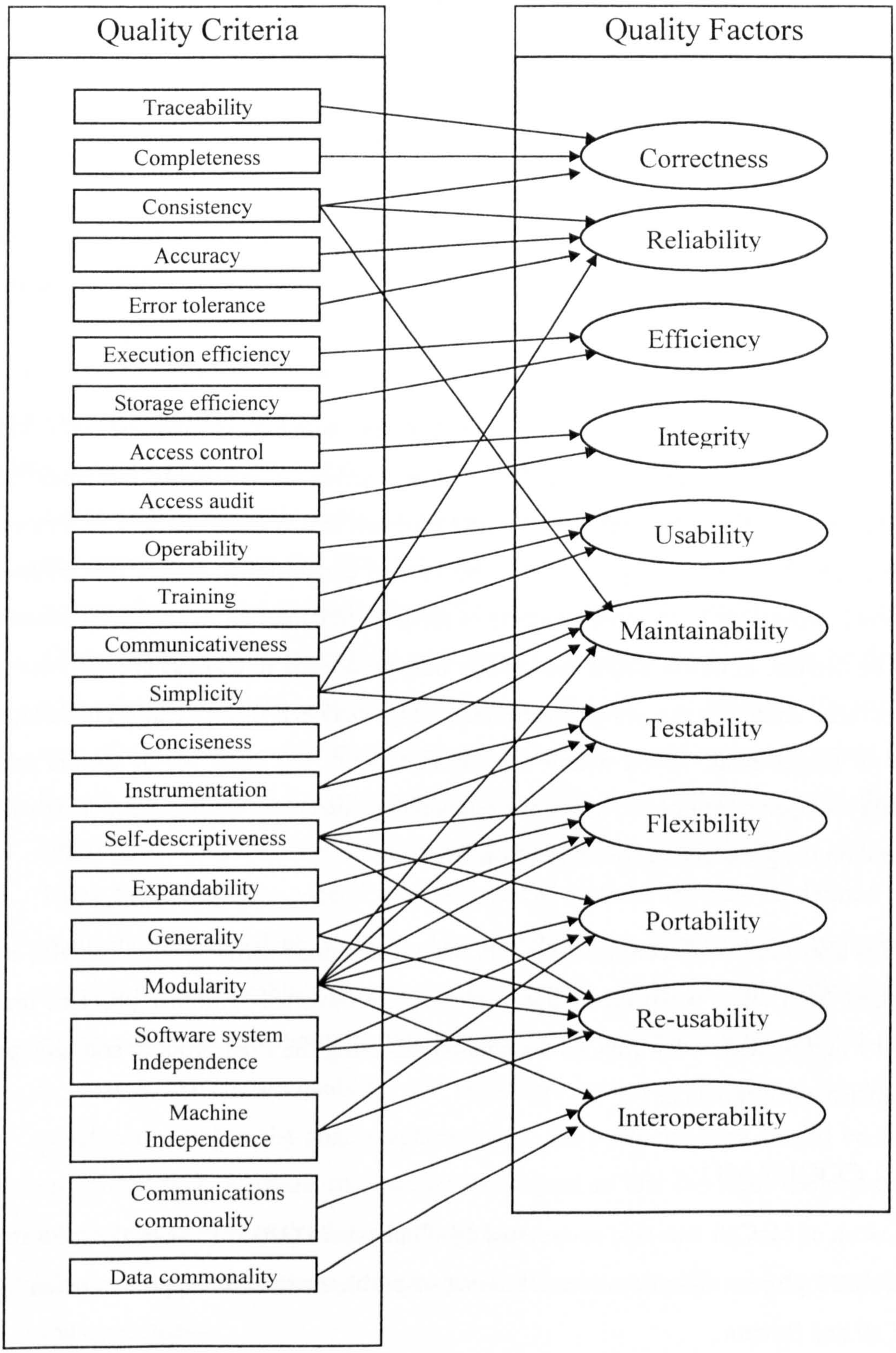


Figure 2-2 How McCall's quality criteria influence his quality factors

Each of the criteria is in turn evaluated on a scale of 1-10 using a set of pre-determined subjective measures. Each quality factor is evaluated using the following equation:

$$F_q = c_1 * m_1 + c_2 * m_2 + c_3 * m_3 + ... + c_n * m_n$$

where, F_q is a specific quality factor, $c_1...c_n$ are the predetermined regression coefficients and $m_1...m_n$ are the values given to the criteria associated with this quality factor.

Although McCall's work was useful in drawing researchers' interest into the quantification of software quality, there are a number of criticisms that may be levelled against the FCM method. The first problem is that the fixed sets of factors, criteria, and the relationships between the factors and criteria, makes the method extremely inflexible. For some products or environments not all of the given criteria may be needed, in others additional criteria may be needed to address special issues, in yet other situations the pre-defined relationships between factors and relationships may no longer hold. Even within a set environment, the exact set of factors and criteria are open to argument, as are the relationships that exist among them. This will depend on the goals and perspective of the assessor.

FCM is also a customer-centred approach, but as mentioned earlier quality can also be assessed from other perspectives and these are not covered. Another problem that may be highlighted is that McCall focuses on assessing the final product and ignores the intermediate products.

2.2.2 COQUAMO

The work of McCall was also re-assessed by Kitchenham (1987) in work done for the REQUEST project. She describes a number of problems with the approach taken by McCall and Boehm:

- There are overlaps between the various quality factors.
- The simple model of factor-criteria-metric obscures the fact that the nature of the

factors, criteria and metrics are very different for different qualities.

- There is no clear indication of the trade-off relationships between factors.
- There is no explicit relationship between criteria, metrics and the life-cycle.
- There is no objective rationale for including, or excluding, a particular quality factor.
- The quality factors are not defined in measurable terms so validation of the postulated relationships between quality factors and quality metrics is difficult.

To help solve these problems she proposed the constructive quality model (COQUAMO). The different natures of the quality factors are dealt with by classifying the factors into one of four categories.

1. Those factors that only apply to certain specialist types of system, i.e., integrity, and generality. Since this type of factor corresponds to the features of a particular system, they are best supported by facility checklists rather than quality metrics.
2. Those factors that are general to most types of system but which must be defined with respect to a particular application, i.e., efficiency and usability (in terms of ease of use). The search for general quality criteria would be inappropriate if the factors are redefined for each application. This type of factor should then be approached using the technique of quality specification (Gilb 1986).
3. Those factors that are general to most types of systems and may be defined in an application-independent manner, i.e., reliability, maintainability, re-usability, extendibility and usability (in terms of acceptance). These general qualities are all capable of direct measurement, but only during the active life of the product. Thus to ensure that these qualities are built into systems, techniques should be developed which assist quality achievement. Software qualities should be cross-referenced with software engineering techniques so that the right techniques are used to achieve the desired quality factors.
4. Those factors that relate to software production, i.e., testability, understandability and correctness.

Kitchenham proposes the following alternative breakdown of quality factors. This model includes an explicit relationship between quality factors and the life-cycle. The diagram shows that each quality factor has a separate checklist for each stage of the life-cycle. These checklists are made up of standards that must be met, metrics to be calculated, and procedures that should be used at that stage to help build-in quality.

Although Kitchenham highlights a number of problems with FCM and suggests an alternative model that alleviates these problems, COQUAMO itself may be criticised on several points.

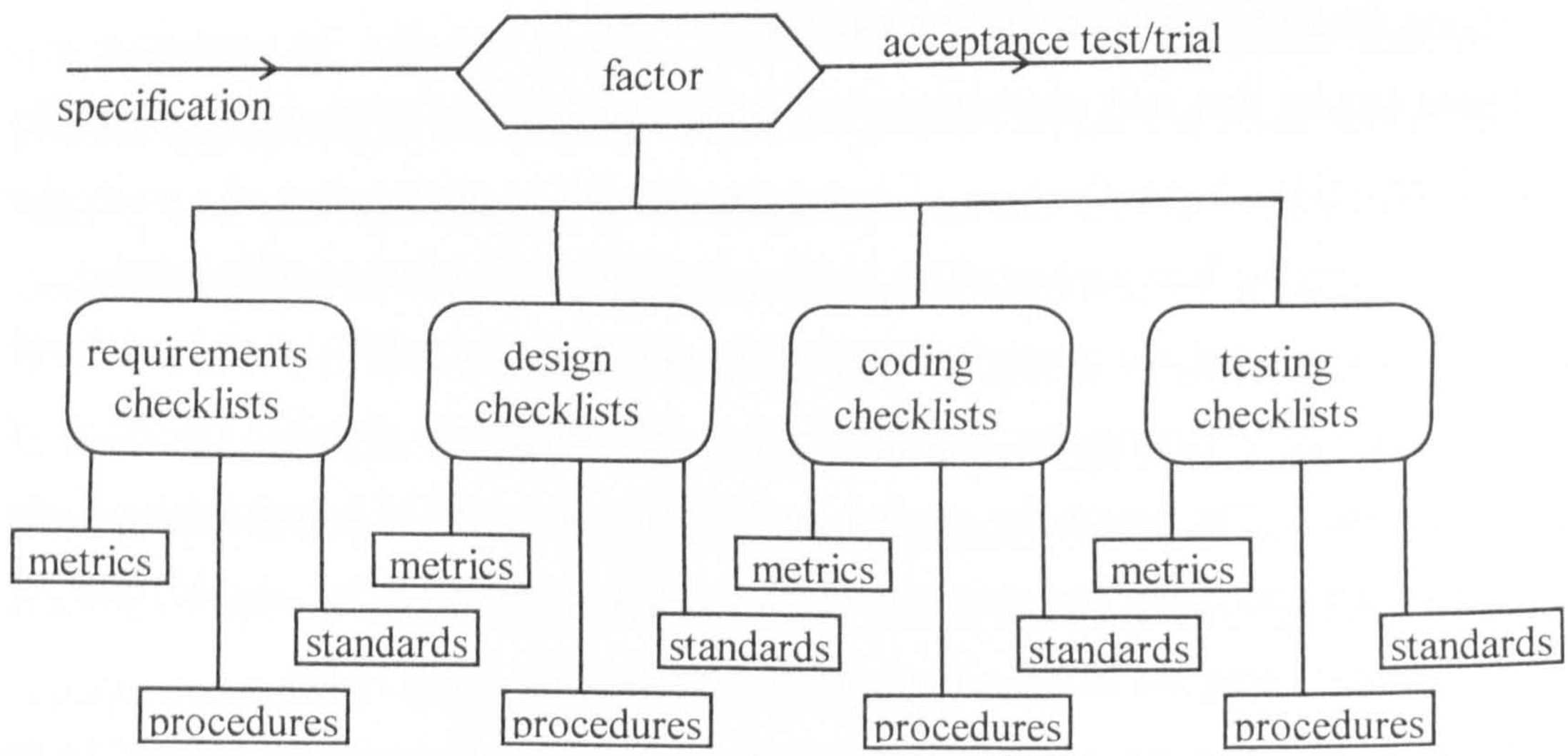


Figure 2-3 Kitchenham's model of quality factors, criteria, metrics and their relationship (Kitchenham 1987)

The first point to consider are the criticisms Kitchenham makes of McCall. The first criticism is that there are overlaps between the various quality factors. Although Kitchenham tries to minimise these overlaps, they cannot be eliminated. One reason for this is that, since several factors share criteria, the factors cannot be considered independent. A second reason is that most of the factors are related to human understanding of, and problem solving with, software artefacts. This means that most factors are dependent upon human understanding and so again cannot be considered truly independent.

Although Kitchenham points out that McCall gives no clear indication of the trade-off relationships between factors, COQUAMO also lacks this indication of trade-offs. Kitchenham adds separate checklists for each lifecycle activity to correct for McCall's failure to give an explicit relationship between criteria, metrics and the lifecycle. However, there is no indication of the relationship between the successive checklists and no clear process for the evolution of the checklists as the lifecycle moves toward coding and more detailed information is available. McCall was also criticised for having no objective rationale for including, or excluding, particular quality factors. However, other than trying to eliminate unnecessary overlap, Kitchenham also has this failing (but she does acknowledge this).

Two of the other criticisms are that the nature of the factors are very different for different qualities, and that the quality factors are not defined in measurable terms. Although both of these criticisms would appear to be valid, Kitchenham's solution to these problems seems inconsistent. She groups the quality factors into four categories: application specific qualities; general qualities requiring application-specific definition; general qualities with application-independent definitions; qualities related to the software production process. There seems no reason to separate process-related measures from the other measures as the difference is in how they are used and not how they are measured.

The first grouping is 'application-specific qualities'. Kitchenham suggests that these factors be best supported by checklists rather than quality metrics. Why shouldn't these qualities simply be either included or left out for a particular project? If they are included, there is no reason why they shouldn't be assessed using quality metrics as for the other qualities. The other groupings are distinguished by the generality of the qualities being measured. The boundaries between these groups are rather arbitrary and artificial. Although it may appear when discussing quality factors at a high level that they may be application-independent, as Kitchenham herself points out, they must be defined in measurable terms. Once an attempt is made to produce this operational definition of factors it is found that the detail of what must be measured depends on

the design methods, programming languages, or document standards used. This means that, even for factors that seem generally applicable, any operationally useful definition is highly dependent on the specific environment in which the software is being developed. The result of this is the conclusion that all qualities to be measured should be defined on an environment by environment basis. This operational definition could be done for all qualities using Gilb's technique of quality specification.

2.2.3 Gilb's quality specification

The idea behind Gilb's quality specifications is that all quality attributes of interest are clearly and explicitly defined at the beginning of the project. These definitions must allow direct and repeatable measurement. They must have an explicitly stated object on which they are measured and a viewpoint from which the assessment is to be made. An example of a definition for 'consistency' taken from (Shepperd 1995) is shown in figure 2-4.

Consistency definition:

Quality Attribute = consistency
Object = information system
Perspective = customer
Scale = probability of a data element being consistent with all other elements in the system.
Test = 1000 random record samples checked by the database consistency testing program
Now = 85-90% (estimated)
Minimum = 90%
Target (11:94) = 99.9%

Figure 2-4 A quality definition framework (Shepperd 1995)

From this example, a number of other aspects of the framework can be seen. The framework is also used to give an indication of the current position, minimum standard and target for the quality attribute being defined. As can be seen from the 'Now' data, values should be given with an indication of their accuracy and degree of confidence. Estimates must be explicitly stated as estimates.

This method does allow for the precise definition of quality attributes in measurable form but it does have a number of weaknesses. The first problem is that in trying to define an attribute in a directly measurable form, the definition of the attribute may no longer completely and accurately reflect our implicit idea of the meaning of the attribute. Since targets and standards are defined in terms of the measurable definition of the attribute, there is a tendency for the definition to 'become' the attribute, rather than describing the attribute. Consequently, measurement processes often optimise the numbers rather than the product.

Another problem is that defining attributes in this way leads to an overly simplistic view of quality (Shepperd 1995). The attributes that a manager may need to assess are rarely independent and these trade-offs need to be taken into consideration. This examination of the interdependencies between a set of attributes can be shown by either a quality factor trade-off matrix or the quality factor deployment (QFD) matrix.

A simple example of a quality factor trade-off matrix is shown in Figure 2-5 (taken from (Shepperd 1995)). This figure shows four quality factors, their priorities and their inter-relationships. A '0' in the matrix indicates no direct relation between factors, e.g., usability and security. Where two factors supported each other a '+ve' would be shown. Usability and timeliness are shown as having a '-ve' relationship. This means that the two factors trade-off against each other. Since timeliness is prioritised as essential and usability only desirable, it is likely that in this case usability would be sacrificed in favour of timeliness.

Quality factor	Priority	Timeliness	Security	Usability
Timeliness	Essential			
Security	Essential	-ve		
Usability	Desirable	-ve	0	
Maintainability	V. Desirable	-ve	0	0

Figure 2-5 Quality factor trade-off matrix (Shepperd 1995)

2.2.4 Quality factor deployment (QFD)

QFD (Kogure and Akao 1983) extends the idea of a quality factor trade-off matrix. It relates the customer requirements with the design requirements that will implement them and gives the strength of the relation.

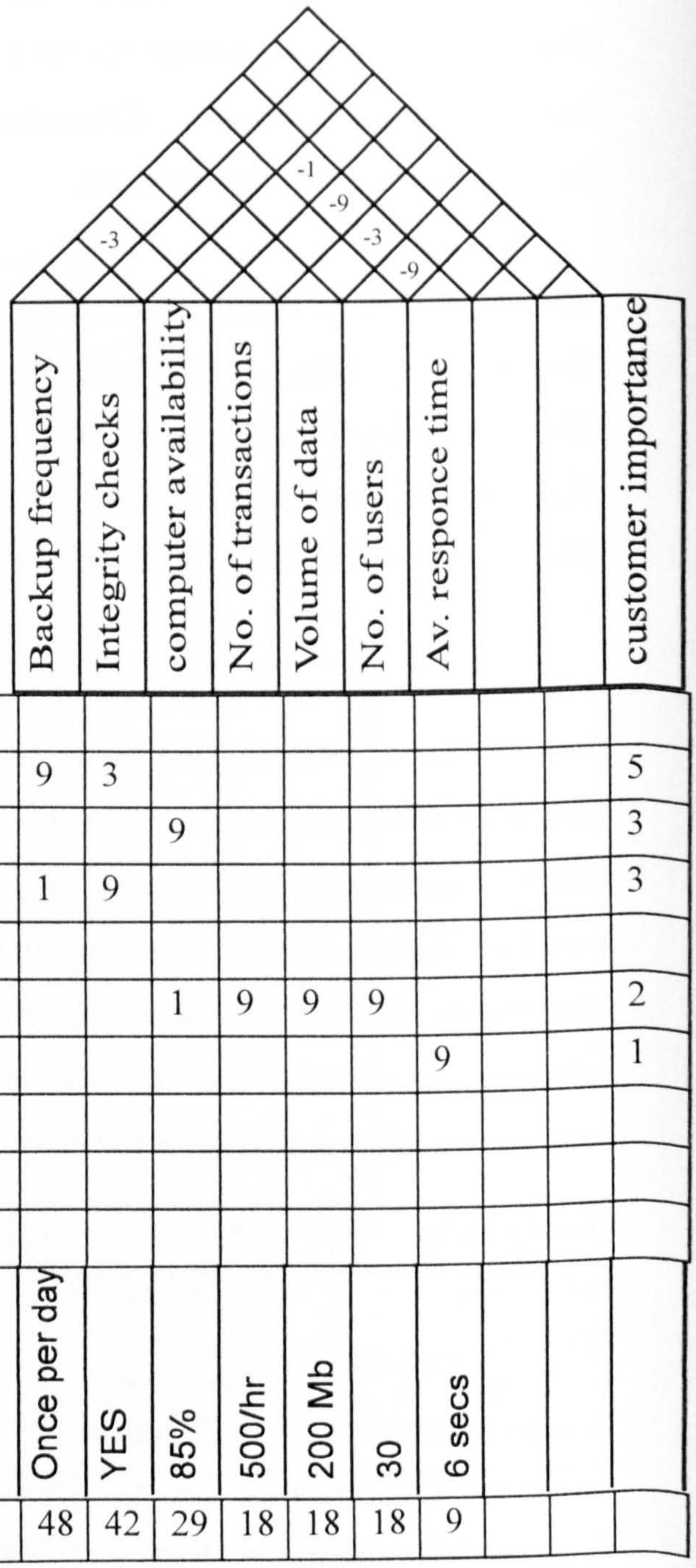


Figure 2-6 An example of a QFD matrix (Shepperd 1995)

The strength of the relation is shown in the matrix by the numeric weight associated with one of the following categories: strong (9); medium (3); weak (1); none (0). Each of the customer requirements is also given a weighting value to show its rank in increasing order of importance to the customer. The values in the relationship matrix are multiplied by the customer weight and summed for each design requirement. This value represents the technical importance rating of each design requirement.The dependencies within the set of design requirements are shown by the correlation matrix in the 'roof' of the diagram (figure 2-6). Again the numeric weight associated with the dependency is one of the following categories: strong (9); medium (3); weak (1); none (0). Since dependencies between two design requirements may be either supportive or a hindrance, positive and negative values are allowed for the correlation matrix. As with Gilb, the desired values of the design requirements are explicitly shown and their units given. This is shown in the 'how much' section.

There are for QFD, as for the other metric methods, some weaknesses. Firstly, by taking a customer viewpoint the method focuses on the quality of the delivered software and ignores intermediate products. As Shepperd (1995) notes 'this provides little scope for remedial action'. This customer centred view also ignores any quality issues that are not apparent in the operation of the final product, such as reuse. In theoretical terms, the association of numeric values with the ordinal scale attributes (used to represent the various relationships within the matrix) and the way they are used in the calculation of the technical importance ratings is somewhat questionable. In QFD's favour, it is more flexible than FCM, in that it allows the definition of relationships and factors on a per application basis.

2.2.5 Goal question metric paradigm (GQM)

Although the COQUAMO and FCM methods are largely concerned with planning, they are generally weak at deciding which quality factors to assess in the first place. QFD takes its quality requirements from the customer (who must also have some means of deciding what they want measured) but this neglects producer quality concerns other than those that directly affect the customer. A method often used to define the measurements which should be taken is the goal question metric (GQM)

paradigm (Basili and Rombach 1988; Rombach and Basili 1990; Basili, Caldiera et al. 1994). This method starts with the high level goals which the measurement program is seeking to fulfil. From these high-level goals a set of questions are developed whose answers will help show whether the goals are being fulfilled. Finally, a set of metrics is produced whose values provide the answers to these questions. Figure 2-7 shows the structure of a GQM hierarchy.

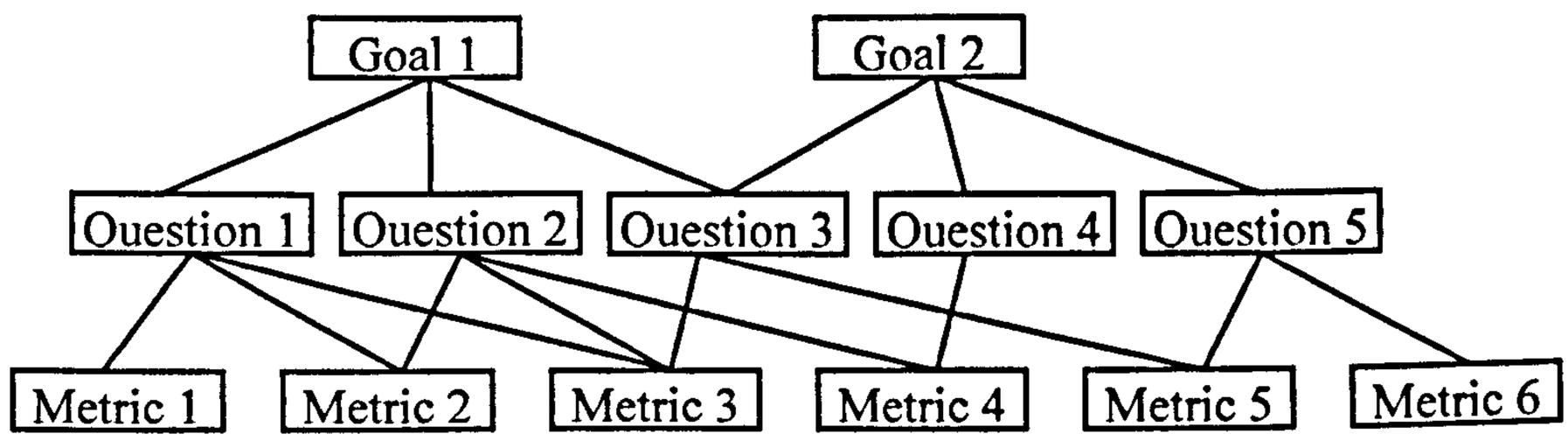


Figure 2-7 A GQM hierarchy

GQM has the strength that all metrics it generates must directly support a high level goal of the software production. It also has the flexibility to be able to measure intermediate products. Although an influential method, GQM still has a number of problems. There is little guidance given in incorporating GQM into the development process. It has been argued that the top-down nature of GQM can lead to a substantial amount of backtracking and re-work if the analysis of data leads to a redefinition of the goals of measurement (Shepperd 1995).

Shepperd also suggests that GQM can lead to a proliferation of metrics. He cites a case at Hewlett Packard (Grady 1987) in which the use of GQM led to the definition of 35 distinct metrics and states that this is 'clearly too many'. This seems a bit of a sweeping statement without any reasons being given as to why this is too many, or what problems having this number of metrics might cause. In a situation where manual collection and processing of data was necessary then the cost of this level of measurement may be unjustifiable. In a situation where metrics were collected and processed automatically this would not be the case. So long as the metrics which are being collected are related to project goals, there seems no reason why 35 metrics or even more might not be usefully and effectively collected. Having said this, there does seem to be a tendency for GQM to produce large numbers of questions and

metrics. Strategies for dealing with this include pruning the GQM tree and a staged introduction of measures on a goal-by-goal basis.

2.3 Quality assessment for OO design

2.3.1 What is quality in design?

Quality for software is usually discussed in terms of the final product. The purpose of the design is to lay the foundations for the final system. This makes it difficult to consider the quality of the design in isolation from the system produced from that design. Although a good design is necessary for a quality final system, a good design needn't always result in a quality final system. If the code is not implemented well, a poor quality system could still result from a good design.

A good design is one that can easily lead to the production of a quality final product. However, this does raise a number of questions. What attributes of a design lead to a quality final product? Can the quality of the final product be estimated based on the information available in the design? How does the assessment of designs fit into the overall development process? The answers to these questions are largely dependent on the information available in the particular notation and on the design process being followed. As this thesis focuses on object-oriented design, a closer look at OO design and the OOD process is worthwhile.

2.3.2 Object-oriented design

2.3.2.1 Elements of a design method

Budgen (1994) lists the three main components of a design method as representation, process and heuristics. The representation can be further split between the concepts being represented and the notation used for the representation (Webster 1996). The notation is the syntax in which the design products are expressed. The concepts are those ideas that it is possible to express using the method. The process describes the sequence of activities performed to produce a design. Pragmatic guidelines to aid in the construction and evaluation of a design are captured by the set of heuristics.

Concepts are largely the same across modern OO methods (as evidenced by the recent successful unification efforts on UML (Booch 1996b) and OPEN (Henderson-Sellers and Graham 1996; Henderson-Sellers, Graham et al. 1997)). The notation in which they are expressed should have little impact on design assessment. The heuristics that accompany a method have a clear bearing on design assessment and this will be discussed later in the chapter. The design process effects design assessment, in an indirect manner, through the assessment activities included in that process.

2.3.2.2 The OO design process

Although the products of design may be diagrammatic, producing diagrams is not designing. As Booch (1994) points out "designing is not the act of drawing a diagram: a diagram simply captures a design". He continues "if you follow the work of any engineer ... you will soon realise that the one and only place that a design is conceived is in the mind of the designer." What are the mental processes undertaken during design? Curtis, Kellner et al. (1992) describes software design as "a collection of interleaved, iterative, loosely-ordered processes under opportunistic control..." and states that "good designers work at multiple levels of abstraction and detail simultaneously". The OPEN method describes the design activity as "perhaps the most loosely defined of the object-oriented lifecycle activities... and is essentially a creative, not a mechanistic process" (Henderson-Sellers 1996b). In their book describing the Syntropy method, Cook and Daniels (1994) describe the design process as "informal and organic". This gives a picture of a somewhat chaotic and unpredictable design process. How is control possible within such an unstructured process?

Cook and Daniels state that "it is necessary to separate clearly the software management process from the software design process, so that designers are not constrained by the formality needed by managers." They give the model of design shown in figure 2-8. In this model, the formal management process establishes "milestones that correspond to the delivery of functioning code and achieve them in a series of evolutionary cycles." This type of management control based on functioning code makes the separation of design and coding processes almost impossible.

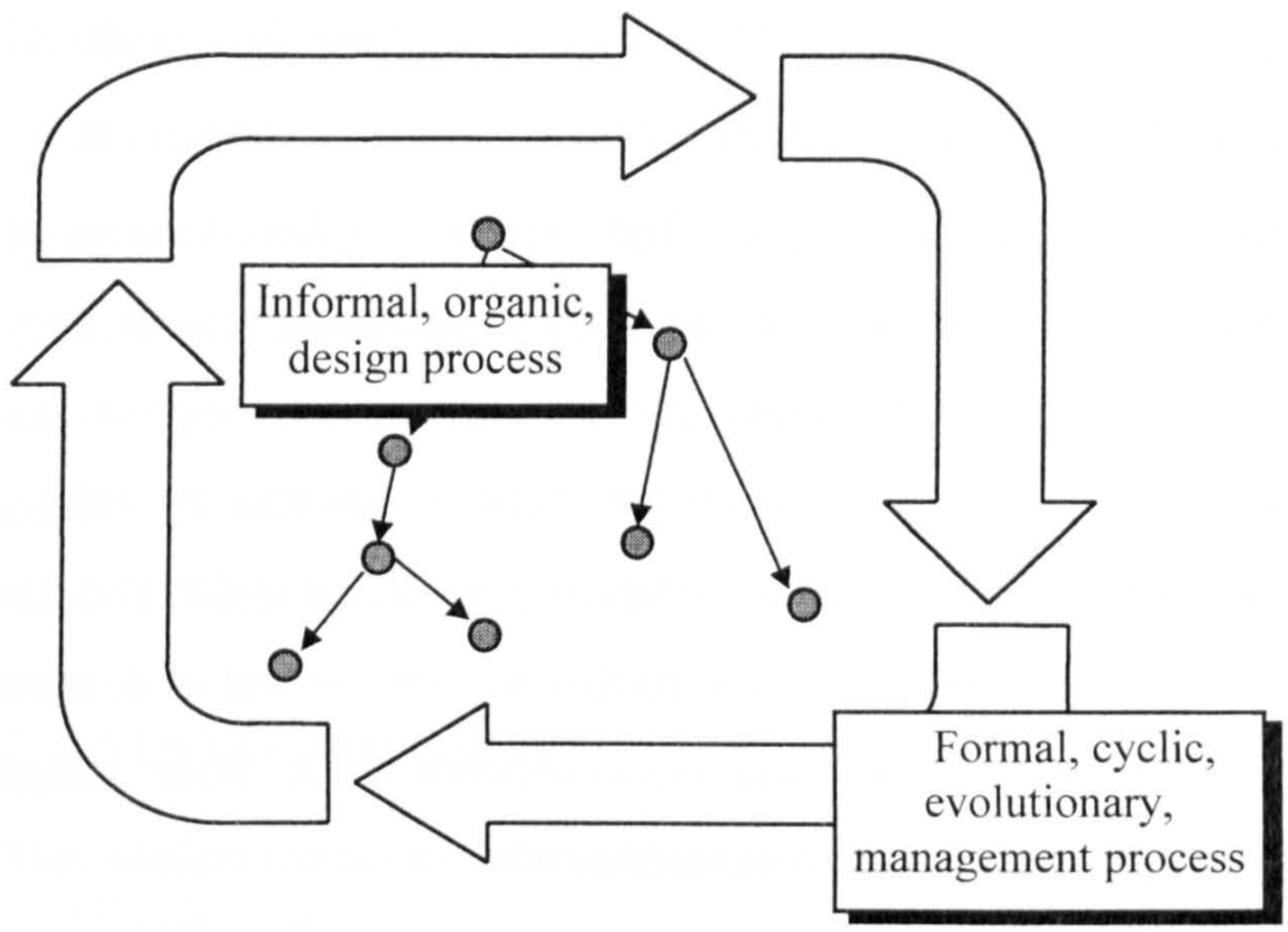


Figure 2-8 Management and design process (Cook and Daniels 1994)

The 'design a little, code a little' incremental approach to object system construction means that designs are often not completed in one go. Instead a high-level architectural design is done and then the more detailed design is done for the classes and subsystems involved in the various increments before they are coded. Those classes and subsystems required for the next increment, but not yet built, are then designed.

Within the informal organic design process performed by individual designers, there is scope for the comparative assessment of designs. The process of design at this level is about considering alternatives. Would one way of structuring a system be better than this other way? If a particular change were made, would the 'new' design be better than the old one?

Assessment within the formal management process is much more of a stand-alone assessment. The questions are, whether there are any problems with this design, or whether this design is good enough. At this higher level process assessment is likely to take the form of design reviews.

Another issue with object-oriented design is the possible lack of completeness of the

designs. According to Cook and Daniels (1994), the "completeness of design models is not always a reasonable or useful goal". They argue that there is a trade-off between the total effort spent on a system and the percentage of that time spent on design. If too little time is spent on design, effort will be higher because of backtracking and reworking. On the other hand, there are diminishing returns from spending a higher and higher percentage of project time on design. At some point, the extra effort spent on design will have insufficient pay back in savings on coding effort. This tends to mean that systems are not as completely specified at design-time as the notation would allow (Martin 1995). This means that any assessment of designs may have to be made on incomplete data and measurements made from design information may not match equivalent coding measurements.

2.3.3 What is quality in object-oriented design?

When discussing object-oriented designs, consideration needs to be given to whether the quality factors and criteria given for systems in general still apply under the object-oriented paradigm. Another question is whether OO requires any factors or criteria that are not applied to structured designs.

Fenton and Pfleeger (1996) describe quality factors as 'high-level external attributes'. Being external attributes, quality factors should remain unchanged whatever the paradigm, method or tool set used to produce internal design of the software. This implies that object-orientation should strive to achieve the same quality factors as any other paradigm. Some quality criteria are more tightly bound than quality factors with the internal structure of software, and so are more likely to be affected by a paradigm shift. Such criteria as modularity and data commonality do not have quite the same meaning in OO design as for structured design. However, many of the criteria would appear to have the same, or very similar, meaning for both OO and structured systems.

Significant differences between OO and non-OO systems appear when the level of metrics is reached. The differences in metrics are reflected in the need felt by researchers to design new metrics specifically for OO systems, rather than simply re-using those which already existed for structured systems.

Much of the discussion presented so far on quality deals with the top-down decomposition of quality. The methods described have started from high-level aspects of quality (factors) that are then hierarchically decomposed until a set of directly assessable items are reached. There is an alternative approach. Experience can be used to identify individual rules or guidelines that indicate quality. These quality guidelines can then be use to assess quality in a constructive, bottom-up manner. These quality guidelines are alternatively known as heuristics.

2.3.4 OO design heuristics

This section introduces the subject of design heuristics. Firstly, it looks briefly at the general idea of design heuristics. A description of a survey and review of the object-oriented design heuristics literature is presented. The work done in producing this survey forms the basis for much of the following discussion and is documented in a technical report by the author (Kirsopp 2001b).

2.3.4.1 What are design heuristics?

Experts in any field appear to act as if there were a collection of unwritten rules that they were following. OO design heuristics have emerged as an attempt to codify OO design expertise. The heuristics take the form of rules of thumb that can guide designers as they choose between various alternatives. They are separately gleaned pieces of distilled design experience that can be shared. Heuristics are pieces of advice, on detailed aspects of design, which are held to be true in most cases (at least by their originators). These partial definitions begin to capture what design heuristics are about, but the topic is broad and lacks a precise definition.

2.3.4.2 The heuristics survey

The survey began as an attempt to gain an idea from the OO literature of how to assess OO designs. It finished as a large-scale cataloguing and analysis operation of OO design heuristics. Not all design advice falls into the heuristics category. Design patterns are pieces of advice based on design experience. They are recommended forms for the solution of known recurring design problems. Patterns are not intended

as a means of design assessment and therefore were not included in the survey (a more in-depth discussion of the relationship between heuristics and patterns will be presented in section 2.3.4.6).

The survey started by looking at the guidelines provided with particular OO methods. General bibliographic sources for the subject area (e.g., ISI scientific citation index, INSPEC) were then searched for both general sets of guidelines and heuristics on particular aspects of OO design (inheritance, aggregation etc.). The survey looked at 71 different sources of heuristic information and found 288 suggestions for heuristics. Although this was by no means a small survey, it is almost certainly incomplete. There are a very large number of publications on OO design, many of which make suggestions on desirable structures or properties for designs. It is hoped that the survey was large enough to be representative of the ideas on the topic of design heuristics from the literature as a whole, as well as capture the majority of distinct heuristic ideas (if not all of their various incarnations).

Many authors present heuristics that appear to be describing the same underlying idea, although the exact definition of terms or choice of thresholds may vary slightly. Clearly, there is some degree of subjective judgement involved in deciding whether two heuristics are sufficiently similar that they can be treated as the same. However, the author found that the 288 heuristics from the survey could be replaced by a list of less than half that number with the minimum loss of meaning. If a 'capture-recapture' view is taken of this overlap, it suggests that a good coverage of the available heuristic ideas was indeed attained.

2.3.4.3 Where do design heuristics come from?

Object oriented design heuristics have arisen in a number of ways, although they have not always been labelled as heuristics. Sets of general recommendations have accompanied design methods, e.g., (Coad and Yourdon 1991b; Booch 1991). Guidelines on specific topics have come out of narrowly focused pieces of research, e.g., an experimentally derived heuristic for inheritance depth (Daly, Brooks et al. 1996), or a heuristic for abstract classes based on theoretical arguments (Hursch

1994). Experience-based sets of heuristics have been proposed independently of any particular design method, e.g., (Coplien 1992; Riel 1996). There has also been the emergence of empirically based heuristics, in which measurement values falling outside of a statistically derived 'normal' range are deemed to be violations (Abreu, Goulao et al. 1995).

2.3.4.4 Heuristic assessment types

There appear to be two distinct ways in which heuristics can be used in assessment. These are stand-alone assessment and comparative assessment.

2.3.4.4.1.1 Stand-alone assessment

In stand-alone assessment, a single version of a single entity is evaluated in isolation. Firesmith (1995) provides an example of a heuristic for stand-alone assessment, 'avoid inheritance structures that are too shallow (3) or too deep (7)'. Here the heuristic contains its own threshold values so that no other entity is required for comparison. As well as these 'threshold' heuristics, there are heuristics that embody rules. For instance, 'subclasses should not delete features of their superclasses' (Firesmith 1995). This type of 'rule' heuristic can be considered as a separate type of heuristic, or as a subtype of threshold heuristics (those in which the threshold is breached by a single occurrence of the situation described).

In stand-alone assessment, heuristics operate as tests that are either passed or failed. For example, an inheritance depth of 2 or 8 would fail the above threshold heuristic, but a depth of 5 would pass. Any classes that deleted any features of their superclasses would fail the rule-based heuristic given above.

Design comparison is distinct from design assessment because it deals with choosing between a set of possible designs rather than evaluating a single design. For heuristics that have clearly defined thresholds a design is assigned to one of two categories, either the heuristic is broken, or it is not. This means that if one design complies with the heuristic and another design does not, the designs can be ordered based on the heuristic. Clearly, this type of heuristic has a poor discriminatory ability in

comparison, as it is effectively a restricted ordinal scale with only two categories.

2.3.4.4.1.2 Comparative assessment

Comparative assessment uses another entity (or another version of the same entity) as the reference point for the assessment. Some heuristics are more directly suited to comparison because they codify a tendency rather than a threshold, e.g., 'minimise the number of classes with which another class collaborates' (Riel 1996). This type of heuristic is more likely to be useful in discriminating between designs. In this example heuristic, a version of a class that collaborates with 3 other classes is considered better that a version that collaborates with 5. However, nothing is said about absolute quality. There is no implication that 3 is good or that 5 is bad, just that 3 is better than 5. There is also no implication of the degree of difference (either by interval or by ratio).

Of course, a potential weakness of comparative heuristics is that they require at least two versions of the entity under consideration. This means that some activities, such as QA or design reviews, that tend to work with a single design cannot easily utilise comparative heuristics. However, there are a couple of workarounds for this problem (as demonstrated by the subjects in the case study presented in chapter 5). Within-design comparisons can be made by partitioning the design into a number of elements, which can then be compared. It is also possible to covert a comparative heuristic into a stand-alone heuristic by adding an arbitrary threshold.

2.3.4.5 Relationship between heuristic and metrics

The review of heuristics for the catalogue has shown that heuristics may have different styles of assessment. On one extreme, are heuristics whose application requires a value judgement on the part of the designer. These heuristics will be termed semantic heuristics as their application depends upon the meaning of the entity to which they are applied. The example heuristic given below is a semantic heuristic.

An object removed from the context of the immediate application, should, in isolation, still represent a coherent and complete object-oriented concept. (Berard 1993)

On the other extreme of assessment styles, are those heuristics where conformance may be checked purely based on the structure of the design/code. These will be termed syntactic heuristics. The next example is a syntactic heuristic.

Avoid having too many services per class. Each class typically has no more than six or seven public services. (Coad 1991c)

Some heuristics fall between these two classifications. Although they are to some extent measurable, some semantic interpretation is also required. The third example heuristic falls into this hybrid category. The size of a class and its level of coupling can be measured, but whether it has too much control and responsibility still has to be assessed semantically.

Do not create god classes/objects in your system. (Riel 1996)

Metrics are related most naturally to syntactic heuristics. Measurement values may be used to automatically check the conformance of a design to a syntactic heuristic. Heuristics may be created as threshold values on particular metrics (Abreu, Goulao et al. 1995). It is also possible to define metrics specifically to automate the detection of violations of particular heuristics. Where the heuristic is created independently of any particular metric, questions arise as to whether it can be assessed using a metric and if so, which metric would be most suitable. A general heuristic is likely to be phrased in terms of some product attribute and this attribute may be measured using a number of distinct metrics. For example, the heuristic "classes should have the minimum of coupling with other classes" (Love 1991) could be assessed using Chidamber and Kemerer's CBO (Chidamber and Kemerer 1994) or Lorenz and Kidd's 'class coupling' (Lorenz and Kidd 1994).

Heuristics may apply to some syntactic element of the design or to a semantic aspect of design. Although syntactic heuristics may be assessed using objective metrics, semantic heuristics are only likely to be amenable to subjective measurements (e.g., Likert scales)

2.3.4.6 Relationship between heuristic and patterns

Patterns and heuristics are similar in that they are both a means of capturing design experience. The type of experience they capture is however different. Patterns are reusable pieces of micro-architecture that may be applied in particular semantic situations (Gamma, Helm et al. 1994). A pattern specifies the shape a section of the design should take. Heuristics only specify a guideline that particular design entities should conform to, and do not specify any particular design structure to achieve this.

As well as normal 'positive' patterns, so called 'anti-patterns' have been suggested (Brown, Malveau et al. 1998). The idea is that rather than proposing how a standard design task should be achieved, an anti-pattern captures common ways in which the task is done badly.

Riel (1996) has suggested that heuristics might be used to trigger standard transformations between 'a bad design pattern' and 'a good design pattern'. Although the transformation between an anti-pattern and its positive counterpart appears to be a sound method of design improvement, it seems unlikely that design heuristics can be used as a means of triggering such changes. Identification of the anti-pattern is what is required to trigger the transformation, not the violation of a heuristic. The violation of a heuristic cannot be linked to a particular anti-pattern because it is independent of the semantic situation necessary to identify a pattern. This means that a heuristic violation could indicate the presence of any one of a number of anti-patterns, or a problem without any associated pattern.

Riel has suggested that these standard transformations might be used 'to automate the design optimisation process'. A range of possible transformations could be suggested to a designer upon the discovery of a heuristic violation. It should also be possible to automatically transform anti-patterns into good patterns, once the designer has identified the anti-patterns. Neither of these possibilities could be fully automated. They both depend on the semantics of the design and so would require intervention from the designer.

2.3.4.7 Contradictions and trade-offs

It is reasonable to think of heuristics as similar to those other distillations of useful experience - proverbs. Proverbs can appear to be in conflict with one another. For instance, consider the following apparently contradictory examples, 'too many cooks spoil the broth' and 'many hands make light work'. On the surface, each of these proverbs seems to be saying the opposite from the other, one appears to argue that fewer people are better and the other appears to argue that more people are better. However, we know that the key to correctly applying these proverbs is the context. The context for the 'too many cooks' proverb is that of decision-making whereas the context for the 'many hands' proverb is that of performing a large well-understood task. The same problem occurs with design heuristics. As soon as we simplify the advice to make the heuristics short and memorable we lose the connection to the context and risk applying them inappropriately.

When a number of heuristics are applied to a design, occasional contradictions appear. These contradictions require explanation. In their abbreviated 'snappy' form, heuristics are presented without any explicit context and this can lead to contradictions. Where an apparent contradiction appears, consideration of the rationale for each heuristic, and its associated context, will usually show that the two heuristics should not be applied in the same situation.

On some occasions, two opposing heuristics can be applied within the same context and this implies a trade-off. In these cases, each heuristic seeks to improve a different quality factor of the system to the detriment of the other. An explicit design decision must be made about which heuristic will be followed and which ignored. The decision will be based on the strength of the heuristics and on the 'quality profile' desired for the system. "The strength of a heuristic comes from the ramifications of violating it" (Riel 1996). The results of violating some heuristics will clearly be worse than the results of violating others. Where a decision cannot be made on the basis of strength, consideration must given to which 'ilities' the heuristics support and which of these 'ilities' is more desirable for the system under consideration.

2.3.4.8 Validity

Since heuristics are simply rules of thumb and typically without supporting empirical evidence, how do we know that they constitute good advice? A high degree of agreement between the heuristics from different authors might increase confidence in the validity of their heuristics. The heuristics survey highlighted little contradiction between different authors within the heuristics literature. Differences do exist in the choice of particular threshold values, but on the general trends, there appears to be consensus on most issues. However, although for some guidelines there appears to be overwhelming support, it is hard to tell if different authors proposing the same heuristic are doing so independently or are feeding off each other's work. If common guidelines were arrived at independently, it would give a much stronger indication of their validity than if the authors merely agreed with heuristics previously proposed by another writer.

In theory, heuristics may be formally validated using external measurement results. For instance, if heuristic A is supposed to support maintainability then we can compare the maintenance costs of systems in which it is used to systems in which it is not. It would be difficult in practice to design an experiment to allow for all of the possible confounding factors. Many aspects of design not covered by a particular design heuristic can affect maintenance costs. They may also be affected by the coding practices used to implement the design. The processes and personnel used during maintenance can also clearly affect the costs. How can we tell if low maintenance costs are due to conformance with the heuristic or one (or more) of the multitude of other possibilities? In practice, the formal validation of individual heuristics is rarely done. However, Briand, Bunse et al. (2001) have published results from an experiment looking at the application of a set of design guidelines proposed by Coad and Yourdon (1991b). They conclude that following these guidelines makes maintenance tasks significantly easier. This result may not show the validity of individual heuristics, but it does support the idea that following a set of design heuristics should lead to a better design.

2.3.4.9 Application of heuristics

Heuristics application may be divided into two main sections, education and practice. In education, heuristics are used to help inexperienced designers to learn and apply the skills and insights of highly skilled and experienced designers (without having to spend 10 years acquiring them). A study of the use of heuristics in teaching OO design has been performed (Gibbon and Higgins 1996). In this study, heuristics were used as a means to transfer object-oriented design expertise to inexperienced designers. They found this to be a 'resourceful teaching aid'. However, since we are concerned with the use of heuristics during the design process, we will not consider their educational use any further here.

In examining the possible uses of heuristics in industrial practice, one approach would be to consider the use of heuristics within the various development process activities. In general, design heuristics may be applied during any activity in which the design (or subsections thereof) is created, modified or assessed. Sommerville (2001) lists four fundamental activities common to all software processes: specification, design and implementation, validation, and evolution. The specification is performed before the design is created and so design heuristics cannot aid specification. Validation is concerned with the externally observable behaviour of a system, rather than its internal design, and so design heuristics are not helpful for this activity either. Heuristics can however be applied during the other activities.

2.3.4.9.1 Design and implementation

The first question to consider here is, could heuristics be used in the initial formation of a design? Heuristics check the conformance of the design to some rule. If we do not yet have a design how can we check its conformance? This suggests that heuristics can only be applied after we have an initial design. During the initial formation, the design may still be wholly within the mind of the designer. The designer may be applying heuristics subconsciously during design formation, but the explicit inclusion of heuristics in any element of the design process must wait until the design exists in some concrete external form. Once there is an external representation of a design, it can be assessed with the aid of heuristics. Assessment involves

checking the compliance of a single existing design against a set of rules. This involves the application of heuristics in a stand-alone manner as described previously.

After the initial design formation, OO designs are usually developed in an iterative manner. A new iteration of the design may be assessed in a stand-alone manner or in a comparative manner. We may want to know if a change is likely to cause problem with the design, in which case we are assessing the new design. Otherwise, we want to know if the changes improve the design, in which case we are comparing designs. The only heuristics that would apply directly to system refinement would be those used to control the refinement process. Heuristics could be used to decide when to stop the iteration in a refinement process.

People other than the actual designer may use design heuristics. An assessment team may use them during a design review. They may also be included in design standards to be applied as QA. In both of these cases, heuristics will usually be used on a single design through stand-alone assessment.

2.3.4.9.2 Software evolution

There are three different strategies of software evolution: maintenance, architectural transformation, and re-engineering (Sommerville 2001).

Maintenance is further subdivided into corrective, perfective, adaptive and preventative. Corrective maintenance involves changes made to repair software faults. These faults may be in the detail of the code and require no changes to the software's design. Alternatively, the fault may be at a higher-level and require design changes. In this case design heuristics could be applied to assess the changed design. This assessment may be done in either a stand-alone or a comparative manner.

Perfective maintenance involves changes made in response to changed requirements. This will usually require changes to the design and so can employ design heuristics. Since there is a change to the requirements, the functionality of the old system is different from that required from the new. This means that it may not be sensible to

make direct comparisons between the designs and a stand-alone assessment of the new design is more likely.

Adaptive maintenance involves changes to adapt software to a different operating environment. Changes to the operating environment impose a different set of design constraints and so, as with perfective maintenance, direct comparisons are probably inappropriate and stand-alone assessment is likely to be used.

Preventative maintenance involves changes intended to make future maintenance easier, but do not change the system's functionality. In this case an attempt is being made to improve the design whilst keeping the functionality the same. This is a clear case for application of comparative heuristics.

Architectural transformation is moving from one architectural style to another, e.g., centralised to client-server. This is a really an extreme case of adaptive maintenance and so the same arguments apply.

Re-engineering is restructuring without adding functionality. This is an extreme case of preventative maintenance and so those arguments apply.

2.3.4.10 The heuristics catalogue

Having a large number of heuristics describing how designs should be structured, or how the design process should be performed, is only useful if you can select a suitable subset of heuristics for your particular purpose. To use the entire set of heuristics would mean including those irrelevant to the goal or inappropriate to the context. To select a subset of heuristics for a given purpose requires a catalogue of the available heuristics and means of searching the catalogue based on the intended use of the heuristics.

The survey data was used as the basis for a catalogue of object-oriented design heuristics (presented in appendix A). Producing the catalogue was not simply a case of finding heuristics. The heuristics were grouped into 20 categories each related to a

particular OO concept or construct. For example, categories include heuristics for multiple inheritance, polymorphism and message passing. The catalogue also includes additional information about each heuristic, such as their trade-offs, lifecycle applicability and objectivity. This information is intended to improve usefulness of the catalogue to those trying to apply OO design heuristics.

As has been mentioned, there is considerable duplication of heuristics ideas from different authors. One of the tasks involved in producing the final catalogue was to try to remove much of the duplication. Appendix A shows both the initial catalogue, containing all of the heuristic suggestions found in the survey, and summarised set of guidelines. It also includes the rationale for the inclusion/exclusion of particular heuristics and the rationale for 'collapsing' several heuristics into one prototypical version.

2.4 Automation in OO design assessment

This section considers the degree to which software quality can be assessed solely from the design documentation and the extent which this assessment can be automated. As has already been discussed, there is no single quality model applicable in all situations. Therefore, little can be said in general about the assessability and automatability of design assessment. One potential way to further this discussion is to analyse an exemplar quality model. A static model would be the most suitable as it does not vary from system to system. McCall's FCM model is the most widely cited of these models and so it will be used as this exemplar.

The approach will be to examine the degree to which each of the criteria can be assessed from the information available at design time. A further consideration is the manner in which these criteria could be assessed. This assessment may be done through objective automateable measures or through subjective assessment. The definition of the criteria or the metrics used to measure them may also be dependent on the design method used. In addition, a given criterion may only be partially assessable from the information available.

In the following paragraphs, arguments are given for the assessability and automatability of each of the criteria. This information is then summarised in table 2-1.

Traceability is the extent to which a design component can be traced back to the relevant section of the requirements. This is dependent on the design method. In some methods it is explicitly given as part of the method and in others is unavailable or must be inferred.

Completeness is the extent to which the full functional requirements have been achieved. If a methodology is used which includes traceability, or if formal methods are used then this may be done objectively. Otherwise, it can only be done subjectively.

Consistency is the degree to which uniform design and documentation techniques are used throughout the project. This requires a consideration of what may be assessed from a design, obviously consistency across a project cannot be assessed from the design alone.

Accuracy is the correctness and precision of computations and control. This is unlikely be assessable until detailed design or coding. It will not be available at the architectural design stage.

Error tolerance is the degree of damage limitation when the program encounters an error. Again, this is dependent upon the detailed design and coding of the system. Such things as exception handling would not usually be shown on an architectural design.

Execution efficiency is the runtime performance of the program. Again, this cannot be assessed until the code is written. A similar argument can be made for *storage efficiency*.

Access control is the degree to which the software provides control of the access of software and data. Access control is part of the functionality of the system and should be specified in the requirements. It should therefore be available at design time. However, the degree of access control can only discerned by a person reading the specification/design and so must be only subjectively assessable.

Access audit is the degree to which access of the software and data can be audited. A similar argument can be made for access audit as for access control, i.e., access audit will also be subjectively assessable.

Operability is the ease of operation of a program. Ease of operation depends upon the logical structure of the interface, which should be assessable at an architectural design level. It also depends upon the 'look and feel' of the interface, and the systems response times, which would not be available from the architectural design.

Training is the extent to which the software helps new users to learn to use the system. The inclusion of training in the system should be visible at the architectural design stage, but its extent will only be subjectively assessable.

Communicativeness is the degree to which the software provides useful inputs and outputs. As with operability communicativeness is a facet of the user-interface design. User-interface evaluation is largely a subjective task although some objective measures can also be taken.

Simplicity is the ease with which a design may be understood without explanation over and above the standard design documents. This is related to cognitive complexity. Some aspects of simplicity may be measured objectively and other must be assessed subjectively.

Conciseness is usually defined as the use of minimum amounts of code to produce a system of given functionality. Assessing the amount of code that will be produced

from a design is possible, but depends upon subjective assessment of the level of abstraction of classes shown in the design.

Instrumentation is the extent to which a program monitors itself and detects errors where they do occur. This is largely a matter for detailed design and coding, and is unlikely to be visible at the architectural design stage.

Self-descriptiveness is the extent to which a design documents itself. It is a largely subjective attribute and will be dependent on the design method.

Expandability is the ease with which (or extent to which) a design can be extended. This is both dependent upon the structure of the system, which is objectively assessable, and on the semantics of the system which must be subjectively assessed.

Generality is the range of possible uses of design components. Again, this is dependent on both the structure of components and on their semantics.

Modularity is a measure of the independence of design components. The independence of design components should be objectively assessable from the structure of the design.

Software systems independence is the degree to which the design is independent from its software environment, such as operating system or non-standard programming language features. Generally the use of non-standard language features will be determined at the detailed design or coding stages and will not be visible at the architectural design stage. Operating systems may however impose a structure on the architecture of a design that will limit the design's portability. This will depend upon the particular operating system.

Machine independence is the degree to which a design is independent from its hardware environment. There are some aspects of design, such as information hiding or layering that can reduce machine dependence. There may also be clues in the

design that indicate machine dependence, such as the presence of device drivers. Other machine dependencies will only be visible in the detailed design and coding.

Communications commonality is the extent to which standard protocols and interfaces are used. The protocols and interfaces of a system and its components are visible at the architectural design stage and so should be assessable from the design.

Data commonality is the extent to which standard types and data structures have been appropriately used throughout the design. With the use of data hiding, the allocation of particular data structures to components is generally left until the detailed design stage.

Quality Criteria	Objectively Assessable	Subjectively Assessable	Not Assessable
Traceability	X(M)	X(M)	
Completeness	X(M)	X(M)	
Consistency			X
Accuracy			X
Error Tolerance			X
Execution Efficiency			X
Storage Efficiency			X
Access Control		X	
Access Audit		X	
Operability		X(P)	
Training		X(P)	
Communicativeness	X(P)	X(P)	
Simplicity	X	X	
Conciseness	X	X	
Instrumentation			X
Self-Descriptiveness		X(M)	
Expandability	X	X	
Generality	X	X	
Modularity	X		
Software System Independence	X(MP)	X(MP)	
Machine Independence		X(P)	
Communications Commonality	X		
Data Commonality			X

Table 2-1 Assessability of quality criteria from design documentation

Table 2-1 summarises the discussion of the individual criteria. Many of the criteria have some aspects that can be measured (or estimated) by objective means, and other aspects that require subjective assessment. This type of criterion contains marks in both the objective and the subjective columns of the table. Entries followed by '(M)' denote method dependence; (P) denotes criteria that are only partially assessable.

The next table investigates the effect of the assessability of McCall's criteria on the assessability of his quality factors. Figures given in this table are in percentages. For example, 2 of the 3 criteria associated with correctness are assessable and 1 is not. This gives a value of 67% in the assessability column and 33 in the not assessable column. Both of these criteria are considered to be dependent upon the method and so 67% is shown in the method dependent column (meth). Of these two criteria one is considered objective and one subjective hence the 33% figures shown in the objective and subjective columns.

Quality Factor	Level of Assessment				Objectivity of Assessment	
	Assess	Partial Assess.	Not Assess.	Meth Depend	Object	Subjective
Correctness	67	0	33	67	33	33
Reliability	25	0	75	25	12.5	12.5
Efficiency	0	0	100	0	0	0
Integrity	100	0	0	0	50	50
Usability	33	66	0	33	50	50
Maintainability	60	0	40	0	40	20
Testability	75	0	25	25	37.5	37.5
Flexibility	100	0	0	25	50	50
Portability	50	50	0	50	25	50
Re-usability	60	40	0	20	30	50
Interoperability	67	0	33	0	67	0

Table 2-2 Quality factor assessability from architectural design

From table 2-2, it can be seen that none of the factors can be fully assessed at design time in an objective manner. Although flexibility and integrity are 100% assessable, they are also to some extent subjective. This means that they may not be fully assessed using an automated system and some expert judgement must be applied. Efficiency being dependent upon the internals of the classes is not considered

assessable at the architectural design stage. The other factors are all to some extent objectively assessable at design time.

Although other authors suggest slightly different sets of 'ilities' and a particular system may not include all of these factors in its quality profile, McCall's factors can be considered a typical decomposition of software quality. Therefore, the above analysis can be considered as demonstrative of the general level of assessability of software quality at design time. It is interesting to note that 68% of inputs to these quality factors are considered to be assessable from the information contained in the design. This shows the importance of good design and backs the idea that a poor design is not redeemable through good coding. Over half (52.8%) of the aspects of quality assessable at design time are assessable objectively. If you're a pessimist, this means that almost half of design assessment has to be done by expert judgement. On the other hand, if you're optimist, it means that over half of the effort and expense of manual assessment can potentially be saved using suitable tools.

2.5 Summary

Assessment in software systems is generally split into validation, verification and evaluation. Validation is inherently informal and though there are techniques to aid system validation it cannot be included in any automated design assessment. Verification can be done using formal methods but again is generally done in an informal manner. To a large extent, verification is to do with the behaviour of the system, what it does rather than its structure. At the level of detail available within designs, it is rarely possible to do much useful verification. It may occasionally be possible to prove that a design is incorrect but not that it is fully correct. Evaluation deals with the quality of the design, without concern for whether the right system is being built, and assuming that it meets its specification. This chapter has largely dealt with evaluation of object-oriented designs.

A summary of the work on quality appears largely as a succession of problems with trying to define quality when it can be considered at different depths and from multiple viewpoints. When trying to look inside the notion of quality and consider its

constituent parts there is the problem of deciding which factors or criteria to include in any model of quality. Even after a decision is made as to which factors to include they still need to be defined in a directly assessable form. However, these operational definitions may not completely capture our intuitive idea of the criteria in question.

When the quality of designs is considered, the high level of abstraction leads to a number of difficulties. A considerable portion of the criteria that may be used to measure a design can not be fully assessed. Even those aspects of the criteria that may be assessed are not all objectively measurable. This means that at the design stage the quality of a design may only be assessed in a partial, approximate, and not fully automatable manner. The lack of information in earlier lifecycle activities, such as design, means that only approximate assessments and estimates can be made at these stages. However, as the lifecycle progresses more complete information is available. This suggests the adoption of a process in which progressively improving assessments and estimates are taken as the product advances through the lifecycle.

The main conclusion that can be drawn from a general consideration of quality is the difficulty in defining a single definitive quality model that covers all view points, goals and environments. This is clearly shown by the number of shortcomings demonstrated in the attempts to produce such a model. However, having said this, these modelling attempts are not entirely in vain. In trying to consider how to model quality, a better understanding of the nature of quality is gained and this can only help in improving the production of quality software.

An examination of the object-oriented design process shows additional problems caused by the incremental and iterative nature of the OO development process, as well as the possibly incomplete nature of the designs themselves. Consideration must also be given to the potential differences in defining quality models for OO systems.

Assessing the quality of a design is largely done by comparing design elements and structures against the accepted best practice. Both quantitative and qualitative aspects of best practice can be formalised and communicated as design heuristics. They are a

means of packaging design experience for reuse. However, heuristics are not the only form of packaged experience and the distinctions between heuristics and patterns were discussed.

A survey and review of heuristics from the OO literature was described. This survey looked for examples of heuristics from the OO design literature. The heuristics found came from a varied set of sources. Some heuristics came packaged with methodologies, others were presented as experience-based guidelines, and some even came from focused pieces of empirical work.

The application of design heuristics was also discussed. A distinction was made between heuristics that operate on a single entity (stand-alone) and those that compare at least two entities (comparative). A discussion was also presented of the development activities within which OO design heuristics may be used. In general, this is any activity where the design is created, altered or reviewed. Several specific activities were identified, these were: iterative design improvement; formal/informal design review; quality assurance; and most system maintenance activities (but possibly not corrective maintenance).

The relationship between heuristics and metrics was discussed. Metrics provide a means of collecting the information needed to assess a particular design entity (or entities) with respect to a particular heuristic. Where the heuristics are objective, there is the possibility of automating heuristic application using metrics. The relatively small number of trade-offs between the heuristics shown in the survey, and the lack of methodology dependent or language dependent heuristics, also show that objective heuristics are likely to be suitable for automatic application. With little need to balance trade-offs or consider applicability, it should be possible to apply the heuristics largely by rote.

The final section of this chapter looked at the degree to which design evaluation might be automated. To do this an example set of quality criteria was examined to see whether they were collectable from design information, and whether this collection

was automatable. The effects that these findings would have on the higher-level quality factors were also discussed. The findings were that around 2/3 of the quality criteria were assessable at design time. Of these, over 1/2 were potentially automatable, but some of these could only be defined in a method specific way.

The overall conclusion from this chapter must be that: quality should be defined on a system-by-system basis, most quality criteria can be assessed at design time, and most of this assessment can be automated. Heuristics provide a means to perform design assessment and those heuristics that are syntactic in nature may be used for automated assessment.

Blank

In

Original

Chapter 3 Software measurement

This thesis takes a measurement-based approach to object-oriented design assessment. In the previous chapter the background literature to design assessment was examined. This chapter looks at the existing literature on software measurement. The chapter begins with a brief history of OO measures. Section two presents a general introduction to measurement, which discusses how measurement is applied to software systems. The third section looks at the techniques available for measurement validation. The process of measure development and the process of measure use are discussed in section four.

3.1 A brief history of OO measures

This section provides a chronological overview of the development of measures for object-oriented systems. This is intended to provide background to the following measurement discussions by showing the evolution of the ideas in the development of OO measures. It also introduces the metric suites used in the case study (Chidamber and Kemerer, Lorenz and Kidd, and MOOD) and places them in their historical context.

The first attempts to measure OO systems reused existing measures intended for procedural systems. Some researchers (Lee, Liang et al. 1993; Tegarden, Sheetz et al. 1992; Coppick and Cheatham 1992) proposed the use of Halstead's software science measures (Halstead 1977) with OO systems. Cyclomatic complexity (McCabe 1976) was similarly applied to OO systems (Tegarden, Sheetz et al. 1992; Coppick and Cheatham 1992), as was Henry and Kafura's (Henry and Kafura 1981b; Henry and Kafura 1981a) information flow metric (Lee, Liang et al. 1993). These pieces of work have been criticised (Cartwright 1998) for failing to take account of problems identified with these measures when they were applied to structural systems and also for failing to consider whether the models on which the measures were based were still applicable to OO systems.

With the recognition that existing software measures couldn't capture the novel aspects of object-orientation (Keyes 1992; Hopkins 1991) researchers began to construct new measures specifically aimed at capturing features of object-orientation. A large number of such measures have now been proposed. Rather than trying to give an exhaustive discussion of each individual measure, an overview of some of the more influential sets of measures will be given.

A notable early piece of work in this area is the metric suite proposed by Chidamber and Kemerer (1991). They suggest six class-level measures focused on inheritance, inter-class coupling, cohesion and size. These measures are probably still the most heavily cited and thoroughly validated metrics in the OO metric literature, e.g., (Harrison and Nithi 1996; Li and Henry 1993a; Sharble and Cohen 1993; Wei 1992). It should be noted though that this validation has not always produced supportive results, e.g., (Hitz and Montazeri 1996; Churcher and Shepperd 1995a).

Li and Henry make use of the Chidamber and Kemerer metrics and suggest five further measures (Li 1992; Li and Henry 1993a; Li and Henry 1993b; Li, Henry et al. 1995; Li and Henry 1995). Their main purpose is to find predictors of maintenance effort. This was done by building regression models using data from two commercial systems that were written in Classic-Ada.

Rajaraman and Lyu (1992a; 1992b) introduce a set of four coupling measures based on a directed multigraph representation. The measures make a potentially useful distinction between inheritance-related and non-inheritance coupling.

Tegarden, Sheetz et al. (1993) present 32 measures of complexity for OO systems related to coupling and cohesion. The measures are grouped into four different measurement levels, variable level, method level, object level, and system level.

Lorenz and Kidd propose a collection of 27 measures that are presented with quality threshold values (Lorenz 1993; Lorenz and Kidd 1993; Lorenz and Kidd 1994).

These measures cover a variety of measurement levels, from method to system, and a wide range of OO concepts. Data is presented from several commercial projects (mostly Smalltalk) but the thresholds do not appear to be directly derived from this data.

The MOOD metrics are a set of six system level indicators covering information hiding, inheritance, polymorphism and coupling (Abreu, Goulao et al. 1995; Abreu and Melo 1996; Abreu and Carapuca 1994; Abreu 1993). These measures have been presented with both theoretical and empirical validation. They are quite complex in their calculation and have seen some adjustment to their definitions between publications. The complexity of these measures is due mainly to the desire, on the part of the metrics' originators, that they reflect in detail the structures/concepts they seek to represent. This is achieved at some expense in terms of simplicity and understandability.

Hopkins (1994) proposes a set of four interface complexity measures. The measures cover both class interface complexity and method interface complexity. Interface complexity is cited as having a major impact on reuse and maintenance, but no validation relating these interface measures to any external measures of reuse or maintainability is presented.

Graham's SOMA metrics are a collection of measures presented for use within the SOMA development method (Graham 1996; Graham 1995a; Graham 1995b). Many of the measures are taken, or adapted, from previous metric sets including Chidamber and Kemerer, and Lorenz and Kidd.

Whitmire (1997)Whitmire proposes a set of measures grouped into nine categories corresponding to OO design 'ilities'. They cover size, complexity, coupling, sufficiency, completeness, cohesion, primitiveness, similarity, and volatility. These measures are derived from Whitmire's own formal theory of objects (Whitmire 1996) and are presented with measurement theoretic validations. This work is very formal in nature and it is hard to see how anyone without an extensive mathematical

background could follow-on from this work (or even be able to collect these measures).

With the increasing number of measures available in the literature authors like Graham (1996), and Hitz (Hitz and Montazeri 1995a; Hitz 1995; Hitz and Montazeri 1995b) started to appreciate that they no longer had to 'start from scratch' when defining measures. They could borrow and adapt from the work of other authors and so hope to improve on previous measures (or make them suitable for new areas of application). The increasing number of measures has also prompted the analysis of whole sets of existing measures from the literature. Briand et al. have reviewed, and presented frameworks for, measures of coupling (Briand, Daly et al. 1999) and cohesion (Briand, Daly et al. 1998b).

This brief historical view has provided some background and context to following software measurement discussions and introduced the metrics suites to be used in the case study. It shows that there is already a profusion of measures in the literature and this thesis will therefore avoid adding any more. After this historical introduction to the influential metric sets, the focus will now change to the fundamental issues and mechanics of measurement.

3.2 Introduction to measurement

3.2.1 What is measurement?

Shepperd (1995) provides the following definition of measurement 'the process of assigning symbols, usually numbers, to represent an attribute of the entity of interest, by rule'.

There are several points in this definition that merit discussion. Firstly there must be a clearly specified entity that is being measured. This entity may be either an object (for instance a keyboard) or a process (say, typing). Secondly, measures do not operate on the entity as a whole. Entities can only be measured with respect to distinct attributes of the entity. It would be senseless to say that the keyboard is 45.5, or the typing is

60. Likewise, attributes cannot be measured without reference to an entity. To say the length is 45.5, or the speed is 60 would be nonsense. Measurement only makes sense when both the entity being measured and the attribute with respect to which the entity is measured are known. It is also necessary to state the units in which the results are being given. Measurements should take form 'the *keyboard* is 45.5cm in *length*' or 'the *typing speed* was 60 words per minute'.

A second point of note is the use of the term 'symbols'. Most measurement results in the assignment of numbers (as in the above examples). However, this is not always the case. Clothes may be sized as 'small', 'medium' and 'large' rather than being measured in inches or centimetres. The quality of fruit or vegetables may be described by assigning a particular class to them, i.e., class1 or class2.

The process of assignment must also be done 'by rule'. This means a well-defined process of assigning symbols to attributes must be used. Following a defined process allows repeatability of measurement. If the same attribute of the same entity is measured repeatedly, the same value should result (within the limits of measurement accuracy). Assignment by rule also allows consistent measurement. By following the defined process, different people should be able to produce the same result (again allowing for accuracy).

As previously stated, measurement is the process of assigning symbols to represent a particular attribute of an entity. The need to define the process has already been considered, but problems may also result from poor definition of the entity or the attribute. Firstly, an individual entity may be poorly bounded. For instance, projects often have fuzzy start or end points, and may involve cross-project activities. If we wished to assess the cost of a project we would need to give careful consideration to which activities were considered part of the project. Secondly, the set of entities for which a particular measure is applicable may be poorly bounded. Many measures have been suggested for class complexity. Are these measures also applicable to a structure in C++ or an interface in Java? Attributes may also suffer from being ill defined, e.g., complexity. When there is no consensus on the meaning of an attribute

there is little chance of defining a measure for that attribute which will be generally accepted.

Even if we have a completely unambiguous measure, there is sometimes variation in the results of repeated measurements. The causes of this variation may be inherent in the measurement process, or the measurement instrument. For instance, measuring a small object with a rule will only be accurate to around 1mm. Error can be reduced through the redefinition of processes or by using a more accurate instrument. Refining our measurement process by ensuring our eye is directly above the edges of the object when taking the measurement (to reduce parallax) should reduce error. We might also change from using a rule to a set of vernier callipers. Any remaining (non-systematic) errors may be handled by statistical techniques based on the distribution of errors. It is important that potential presence of measurement error is recognised and that those involved in measurement are aware of its possible sources. This will facilitate improved measurement and lead to the correct handling of uncertainty during measurement analysis.

3.2.2 Why measure?

Measurement is done primarily to find (or help to find) the answers to questions. It also helps to remove, or at least reduce, the level of subjectivity in the assessment of attributes. Measurement has the capacity to move software production away from a 'black art' and toward a fully-fledged engineering discipline. For software production to be an engineering discipline the various activities involved must be understood and controlled in a quantitative way. Measurement is necessary for the formation of descriptive models to aid the understanding of software production. In addition, as DeMarco (1982) famously observed, 'you can't control what you can't measure'. This means that measurement is essential in providing feedback on the status of projects to facilitate control. Along with this use of measurement for understanding and control, is their use as the basis for process improvement and quality standards, such as CMM (Paulk, Curtis et al. 1993) and ISO9000 (ISO9001 1987).

3.2.3 Terminology in software measurement

There are many terms used to classify measures and measurement-related concepts. This section is an attempt to clarify the meaning of these terms.

3.2.3.1 Direct / Indirect

This is a classification of the measures themselves. A measure is direct if the measurement can be made without reference to any other attributes or entities (Fenton and Pfleeger 1996). There are no models directly involved in direct measurement, but direct measurement must serve a purpose and so the measure must be part of at least one theory/model (Kyburg 1984). For indirect measures, the value of the desired attribute is computed, based on a model, from the values of other measures (either direct or indirect).

3.2.3.2 Internal / External

This is a classification of the attribute being measured. Internal attributes of an entity are those that can be measured purely in terms of the entity itself (Fenton and Pfleeger 1996). External attributes can only be measured with respect to how the entity relates to its environment. Here the behaviour of the entity is important rather than the entity itself (Fenton and Pfleeger 1996).

3.2.3.3 Objective / Subjective

This is a classification of the measurement process. Subjective measures require personal judgements from those performing the measurement. Objective measures have defined processes that do not require personal judgement. In theory, any objective measure should be able to be automated and so should not require any human intervention for its calculation.

It might be argued that the subject/objective split is more of a continuum than a dichotomisation. In this interpretation an individual measure may be neither wholly subject nor wholly objective, but may be positioned somewhere between these extremes. This view is attractive because some measures seem to be to some extent both subjective and objective. The author's view is that subjective/objective split

should be considered as a dichotomisation. In this view the assessment of any 'atomic concept' will result in either a wholly subjective or wholly objective measurement. However, some measures actually conflate two or more atomic concepts and this is how they can develop a character that is seemingly midway between the extremes.

Some authors suggest that any metrics that require subjective assessment are open to question (Henderson-Sellers 1996a). The belief seems to be that the only good measure is an objective measure. However, some software attributes that need to be assessed are inherently semantic. That is, they depend on the meaning of constructs in the software products/processes. For example, the cohesion of the elements of a class depends upon the entity or concept that the class represents. This assessment of meaning clearly requires human intervention and results in a subjective measure.

The main purpose of the distinction between objective and subjective seems to be one of automation. In theory an objective measure should be suitable for automatic collection. Subjective measures require human intervention and so cannot be fully automated.

3.2.3.4 Product / Resource / Project

This is a classification of the entity type. Fenton and Pfleeger (1996) list three entity types, product, process and resource. Any activity, or collection of activities, performed during software development is a process entity, e.g., maintenance, code inspection. Any artifact that results from a software development activity is classified as a product, e.g., specification document, test harnesses, final code. Entities that are required/used by a process activity are classified as resource entities, e.g., printer paper, programmers.

Whitmire adds a fourth entity type to this list, project. Here a project is considered a particular instantiation of a generalised process. Companies may have a defined process to follow for all software developments, but each individual development is a project. This separation of process and project doesn't seem particularly useful. To allow projects to be a separate type of entity process needs to be downgraded to an

abstraction and might therefore almost be considered a product. Furthermore, Whitmire (1997) states that '... a project is an instance of process, among other things. The reverse isn't always true: An instance of a process does not always coincide with a project'. This leaves the question how do we classify non-project instances of processes? They are clearly not resources or products, yet they are an instance of a process rather than the process itself.

3.2.3.5 Design / Code

Measures are often described as a 'design metric' or a 'code metric'. Here the classification is based on the point in the software development lifecycle appropriate to the entity being measured. The terms are almost exclusively used for product measures. So a design metric is a measure of an artifact produced during the design phase of a development.

The situation is slightly more complicated than the above definition might suggest. The information available in different design notations differs. This means that in some notations there will be sufficient information for a metric to be used on the design. Using a different notation may mean that the same metric cannot be collected until after coding.

3.2.3.6 Theoretical / Empirical

Theoretical and empirical validation may be mistakenly considered as if they were measurement activities. This can lead to difficulties when considering where they fit into measurement processes. Actually, they are techniques that may be applied during several measurement activities.

Theoretical validation is any attempt to test or refute the correctness or representativeness of a measure without recourse to observation. Empirical validation is the process of testing whether the theoretical properties of our measures and models match with reality.

There are problems with this often-used distinction. Measurement theory is the technique most commonly cited as theoretical validation. However, measurement theoretic validation requires an 'empirical relational system'. In other words, a model of the empirical behaviour of an attribute is required prior to a measurement-theoretic validation of a measure for that attribute. It is also the case that empirical studies are dependent on theory for their design and that empirical observation is dependent on theory for its interpretation (Chalmers 1982). These points show that in reality there is no clear-cut distinction between theoretical and empirical validation. Validation techniques will be discussed in more detail in section 3-3.

3.2.3.7 Measure or metric?

Another point of terminology that should be addressed is the use of the words 'measure' and 'metric' within software measurement. Strictly speaking, a 'metric' is a generic distance measure. However, the term has been widely used in the computing literature to denote individual software measures. This means that to most readers the terms metric and measure are interchangeable and so there seems little point in holding to the above distinction. The term 'metrics' is also used to describe the subject area of software measurement.

3.2.4 Measurement and modelling

It has been proposed that it is only within the context of a model or a theory that a measurement has a meaning (Kyburg 1984). Shepperd and Ince (1993) support this view and use the following example to illustrate the point.

> Consider the measurement observation that a piece of code has a luminosity of x. This cannot be interpreted because there is no theory or model to link the measurement with any other software engineering property of code... We cannot say whether a luminosity of x is good or bad. Nor can it be stated whether it will lead to problems of reliability, and so on and so forth.

If we require models to provide context for our measures, what are these models and what form do they take? Shepperd and Ince present their view of a model in the diagram below (figure 3-1).

Models are abstractions from reality (or at least our perceptions of reality). As with all abstractions they selectively include only certain elements from reality. Models are created for a specific purpose and it is this purpose that guides the abstraction process. It helps to decide which elements of the system being modelled are included and which are not.

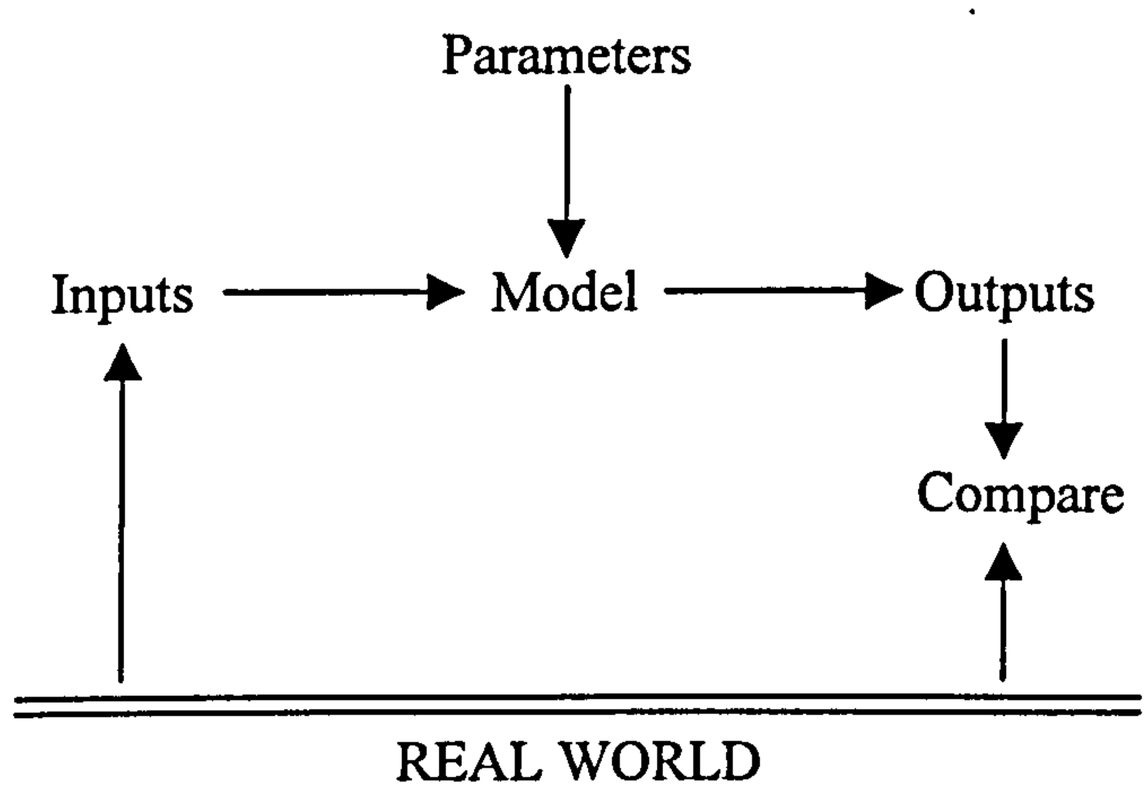


Figure 3-1 The relationship between a model and 'reality' (Shepperd and Ince 1993)

Other important features of a model are the mappings to and from the real world. Inputs to the model must come from the real world, typically though measurement. The outputs from the model are of little use unless they can be interpreted in real world terms. This necessitates that the inputs and outputs of the model have operational definitions. Failure to create mappings to and from the real world creates a 'metaphysical model' (Shepperd and Ince 1993).

The act of abstraction produces a simplified model of reality and as such the model will not behave as the real system would in all situations. Where possible such limitations in the scope and generality of the model should be documented.

Modelling enables the meaningful application of measurement. The explicit use of models enables measures to be properly validated and evaluated. Since modelling should always be done for a defined purpose it helps maintain a focus on the problem at hand. It also gives a clear goal against which measures and models may be

evaluated. One way of thinking about a model is by regarding the relationships within a model as a theory. Expressing these relationships as equations forces rigour into the development of the theory and facilitates validation of the theory and therefore the model.

Shepperd's diagram of modelling shows only those elements in the formal world and not their correspondence to real world entities. It is also unclear where the parameters come from and how they differ from the other inputs. The output of the model is also shown as being compared with something (presumably an external measure) and although this would be correct for validation, other types of interpretation of the result are possible. The diagram shown in figure3-2 is a reworking of Shepperd's diagram of model relationships attempting to address these issues.

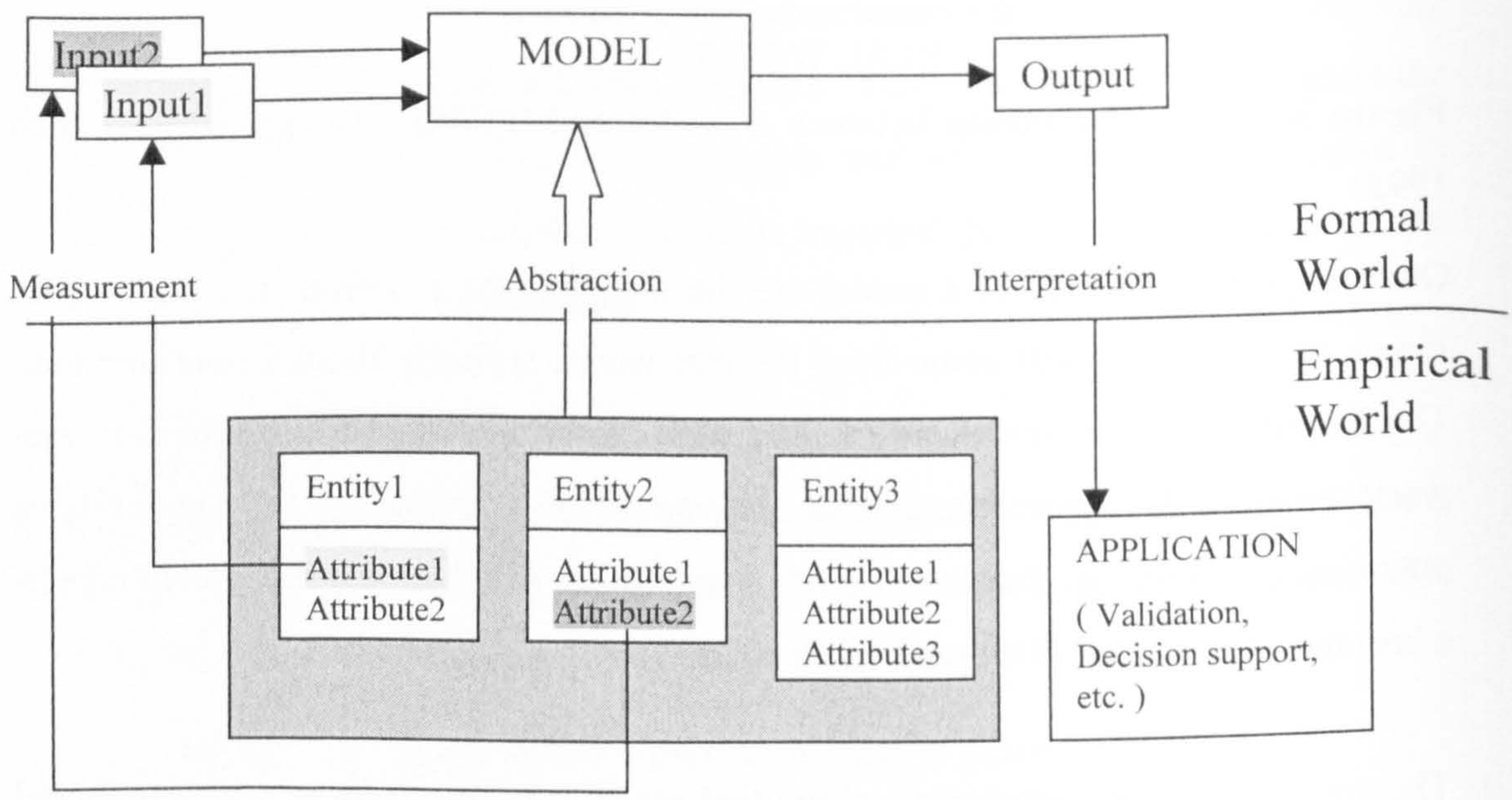


Figure 3-2 Extended view of model relationships

In figure3-2, the inputs to the model are clearly shown as measurements of particular attributes of particular entities in the system being studied. The model itself is shown to be an abstraction of the system. The interpretation of the output has been widened to include activities other than validation.

Although modelling is regarded as essential in measurement, it does not address all of the problems of measurement. Modelling does little to address errors in measurement.

Indeed the simplifications performed during the abstraction process can introduce error into indirect measures and prediction systems. It is not always easy to separate measurement error from modelling error. This means that it becomes more difficult to eradicate (or at least reduce) errors because the cause is not always clear. Another problem that surfaces during modelling is the need for operationalism. Having operational definitions of the attributes being measured is necessary but can lead to problems when the operational definitions are too far from the intuitive idea of those attributes. This type of misplaced operationalism means that, although we now have something quantifiable, it is not the attribute we set out to measure! Chidamber and Kemerer's LCOM metric (Chidamber and Kemerer 1994) may be considered as an example of this problem. Cohesion is a semantic attribute, but by trying to produce an automatable measure of cohesion a structural measure with little relation to our intuitive idea of cohesion was produced.

Kitchenham, Pfleeger et al. (1995) suggest the following types of models that can be built in software measurement:

- Attribute relationship models
  - Definition models
  - Indirect measurement models
  - Prediction systems
- Unit definition models
- Instrumentation models
  - Representational model
  - Theory-based model
- Entity population models

Attribute relationship models embody a system for quantifying one attribute from the values of one or more other measurements. They can be subdivided into definition models, indirect measurement models and prediction systems. The output attribute of a definition model has no external definition, the model is the definition. The main distinction between the other two models is that indirect measurement models must be

based on a causal link between the inputs and the output. In prediction systems a statistical link is sufficient. This distinction can also be seen from their use of units. In an indirect measurement model the result of the equation embodied by the model will produce a result in the correct units for the attribute being indirectly measured without unit correcting constants having to be applied. Where the units of the result do not correspond with the unit of the attribute being quantified, the model embodies a prediction system and not an indirect measure.

Unit definition models define the measurement units for ratio and interval scale measures. Units may be defined with respect to a standard or a theory. It is also possible to define one unit with respect to another unit. This type of model is not generally useful in software measurement because software measures tend to be (or tend to be based-on) simple counts. There may be a case for defining the scale points of categorical measures, though this would not be considered a unit definition in classical measurement.

Instrumentation models describe how the act of measurement takes place, i.e., they describe the measurement instrument. They relate the actual observations made with the result. Simple instruments (e.g., a rule) directly represent the unit used for the measurement. More complex theory-based instruments (e.g., a thermometer) often describe how the attribute of interest manifests itself in a particular entity. In software terms, an instrument model shows the workings of a collection tool or describes the counting rules for manual collection.

Entity population models are used in data analysis or data interpretation. They define the statistical population of the set of entities concerned in our analysis, rather than the actual sample of entities that we are analysing. This allows standard distributions and descriptive statistics to defined for the population. The actual sample of entities that are measured can then be assessed by comparison with this standard. If the population of interest cannot be defined or the entities being studied cannot be shown to be a representative sample of this population, the results from the analysis cannot be generalised beyond the sample.

Kitchenhams's list seems to have been derived by analysis of measurement within the physical sciences. Most of the models described above do not fit the description of a model given earlier in this chapter. Although termed models, most would not be presented in any format suitable for validation (at least for software measures). How would you validate a list of categories, or a set of counting rules? The only models in the set proposed by Kitchenham et al. that would fit the earlier definition of a model, and would be capable of validation, are the attribute relationship models. Software engineering is at best a soft science and approaches to measurement within software engineering should reflect this rather than getting bogged down in 'physics envy'.

3.2.5 Measuring object-oriented systems

This thesis deals with the measurement of object-oriented systems. It would be reasonable to ask, 'what's different about OO systems that they should be considered separately from other systems?'

The argument for OO specific measures is based on the differences between OO and structured software. OO software contains constructs that simply do not exist in structured software, such as, objects, classes, inheritance, polymorphism, and aggregation. Concepts such as coupling and cohesion, while still applied to object-oriented systems, have interpretations that are far more complex. The difference between OO and structured systems are not restricted to the systems built, but also extend to the processes that produced those systems. OO systems are often produced in a more iterative and incremental fashion. Different processes require differences in process control and hence may require different measures to assess progress and aid decision-making.

If only measures designed for use on structured systems are applied to OO system then they will fail to describe many of the important aspects of OO systems. They may also fail to adequately describe some concepts such as coupling whose definition differs between OO and structured systems. The only reasonable solution to these problems is to derive new measures specifically designed to quantify aspects of OO systems.

However, it is also important not to 'throw the baby out with the bath water'. Many measures that were originally designed for procedural systems are equally applicable to OO systems. For example, measures such as function points (IFPUG 1992) that measure requirements should be independent of the implementation technology. Cyclomatic complexity (McCabe 1976) will be as good a measure of method complexity in OO systems as it was of function complexity in procedural systems (although how good that was, is open to question (Fenton and Pfleeger 1996)).

3.3 Measurement validation

Validation is the process of showing measures to be sound and justifiable. As discussed in section 3.2.1, a measure must be of a particular attribute, measured on a particular type of entity, in a given unit and measured using a defined process. These points are necessary for a valid measure but they are not sufficient. The measurement literature contains a number of techniques to help ensure valid measurement.

There are various techniques generally described as theoretical validation. They include, axiomatic / property-based schemes, algebraic validation, dimensional analysis, and the most widely used (or at least cited) of these approaches, measurement theory. Empirical techniques can also be applied to validate measures and measurement related models. A discussion of each technique follows.

3.3.1 Axiomatic / Property-based measurement validation

Many authors have suggested properties of software measures that they consider desirable or essential. Zuse (1997) lists sets of properties from twelve different authors. The wide variation in the sets of properties suggested by these authors shows the variety of intuitive views of measurement still in evidence between the various workers in the field. An example of this disagreement can be seen when examining the literature regarding one of the most commonly cited sets of properties, those proposed by Weyuker (1988) for complexity measures. In their review of measurement validation Kitchenham, Pfleeger et al. (1995) only unreservedly accept one of the nine properties. The remaining eight they either directly refute, or consider simply unnecessary.

These on-going disagreements between researchers, about desirable and undesirable properties of measures, make it difficult to have any confidence in any particular set of properties for measure validation. It also seems unrealistic to expect a single set of universally applicable axioms.

3.3.2 Algebraic validation

In algebraic validation, a formal model of the measure is produced. An axiom set is then defined that describes the desired behaviour for the model. A distinction between this approach and much of the property based schemes, is that the axiom set is produced on a measure by measure basis rather than producing a single axiom set for all measures. The validation itself is done by proving that the axioms are invariant over the model.

Algebraic validation forces an unambiguous definition of a measure's counting rules. It may also be seen as producing a higher degree of confidence than empirical studies. This is because the validity is proven for all valid instances of the model and not merely shown correct for a limited sample. Along with other formal techniques, algebraic validation may find a different set of problems with a measure than those identified by empirical validation (because of the different approach they take). Algebraic validation is not good for checking modelling assumptions or evaluating the scope of models. It also requires a priori knowledge of the model behaviour (Shepperd and Ince 1991), so in this respect it suffers similar problems to axiomatic/property-based validation.

3.3.3 Dimensional analysis

Another potential means of validation, and one that is used extensively in the physical sciences, is dimensional analysis. A measure is invalid if its results have a different dimensional make-up from the attribute being measured. This method of validation has seen few mentions in the software literature because of the difficulty in deciding the fundamental dimensions of software (Henderson-Sellers 1996a). In the physical sciences the fundamental dimensions are well known, and generally agreed. This

makes dimensional analysis a relatively simple check on the validity of models and measures. Until the dimensions of software are understood, dimensional analysis will have little application in software measurement.

3.3.4 Measurement theory

As stated earlier, classical measurement theory is probably the most widely cited form of validation for measures. It is a branch of mathematics/philosophy that is based on the model theory of logic. It outlines how valid measures can be defined and underpins the types of data analysis that are meaningful for particular measures. In the early days of software measurement, measures tended to be proposed based on nothing more than rational argument. This lack of rigour was superseded by the 'strict' application of classical measurement theory to validate new measures. This approach was popularised by Fenton and Zuse (Fenton 1991; Zuse 1994). More recently, there have been moves toward a more pragmatic use of measurement theory, as suggested by Briand, El Emam et al. (1996).

The three main components necessary in defining a measure are an empirical relational system, a representational condition, and a uniqueness condition. The empirical relational system is the set of entities of interest, the empirical relations that hold between them and the set of operations defined on them. The representation theorem states that, a measurement must map an entity's attribute onto a formal representation in such a way that the relations between the empirical world entities must hold between their formal world equivalents. The uniqueness condition is the set of all transformations after which the representation condition still holds. Classical measurement theory uses the uniqueness condition to define the scale of a measure and to classify a measure as belonging to a particular scale type. This scale type is often used to proscribe and prescribe types of analysis for the results of the measure.

The fundamentals of measurement theory are expressed at length in a number of publications. The definitive work on measurement theory is the three-volume set written by Krantz, Luce et al. (1971; 1989; 1990). Other good general works are the

book by Roberts (1979) and an overview paper by Finkelstein and Leaning (1984). There is also a substantial body of work considering the application of measurement theory to software measurement, e.g., (Fenton and Pfleeger 1996; Whitmire 1997; Zuse 1994). For readers unfamiliar with this work there is brief summary in appendix B.

3.3.5 On the application of measurement theory to software

There are some points that may usefully be made concerning the way measurement theory is commonly applied to the area of software measurement.

3.3.5.1 Empirical relations and intuition

The empirical relational system in measurement theory is a representation of our understanding of a particular attribute of a real world entity. The empirical relations describe our intuition of the behaviour of this attribute. As Fenton states 'our intuition is the starting point for all measurement.' In areas such as software measurement where there is considerable uncertainty over the meaning and behaviour of the basic attributes being measured this is a serious problem. If we cannot agree the empirical relational system for an attribute such as complexity, then we have no way of validating any measures of complexity.

3.3.5.2 The prescription/proscription of analysis techniques based on scale type

There has been a tendency in software metrics research toward the strict application of measurement theory as led by Zuse and Fenton, e.g., (Zuse and Bollmann 1989; Zuse 1991; Fenton, Lawrence Pfleeger et al. 1994; Fenton and Pfleeger 1996). Briand, El Emam et al. (1996) have published a critique of this approach suggesting a more pragmatic use of measurement theory in software engineering. The argument is not that measurement theory is of no benefit to software measurement, rather that a less rigid application of the theory might bring greater benefits.

One of the central results of measurement theory is that the allocation of a measure to a scale type allows the prescription/proscription of certain statistical techniques for

analysis of the results of that measure. This is not a view shared by most statisticians and data analysts (Velleman and Wilkinson 1993) for three main reasons.

1. Permissible statistics should be decided by meaningfulness (rather than the representation or uniqueness conditions)
2. The allocation of measures to scale types is often unclear
3. The prescription and proscription of statistics need not be followed as strict rules

3.3.5.2.1 Permissible statistics should be decided by meaningfulness

Velleman and Wilkinson (1993) state that the scale type is not actually an attribute of the data itself. It depends on the use to which the data is put and whatever additional information is available that relates the question being asked of the data. He quotes several examples from different authors supporting this case. Measurement theorists take a different approach in which the scale type is based solely on the data itself (based on the permissible transformations). Neither camp is either right or wrong, they simply approach the problem from different perspectives. Measurement theorists take a mathematical viewpoint in which the ultimate test of validity is internal logical consistency. Statisticians tend to take a scientific viewpoint where validity is tested against experience. Which approach should be taken in software engineering? Software engineering must ultimately be a pragmatic discipline (that's what engineering is about). Therefore, the author believes the approach taken by the data analysts would be the most suitable for software engineering research.

The above argument for meaningfulness clashes with some uses of scale types in software engineering. As an example, Zuse (1997) describes the 'number of methods' (NOM) metric as ordinal without any reference to how it is to be used, based on a strict measurement theoretic approach. Considering several possible uses for NOM shows that it can take on a number of scale types. If we wished to use NOM to find the number of classes in a system which have 5 methods we would be using it as a nominal measure. Using NOM as a surrogate for another size measure will lead to an ordinal use[†]. When trying to find out how much larger classes were after a

[†] Zuse's argument using for ordinal scale is based on composition, this suggests that he is implicitly assuming the use of NOM as a surrogate size measure. Saying that it is not a ratio scale surrogate further suggests that NOM is being used as a surrogate for a 'correct' (but unspecified) size measure.

maintenance change, we would be using it in an interval manner. Finding the mean number of methods per class would treat NOM in a ratio manner. The NOM metric illustrates the fact that a strict adherence to measurement theory would unnecessarily disallow sensible and useful applications of software measures.

3.3.5.2.2 The allocation of measures to scale types is often unclear

As we have already seen the choice of empirical relations is dictated by intuition. The scale type is dependent on the uniqueness condition, which is in turn dependent on the empirical relations. This chain of reasoning means that the selection of scale type is ultimately the direct result of mere intuition and so in many cases we cannot be certain of the scale type of a particular measure.

Another problem in the allocation of a measure to a scale type is demonstrated by the NOM example above. There is clearly an argument that the scale type may not belong to the measure at all, but to the use to which it is put. If this argument is true, any attempt to assign any measure to a scale type may be flawed.

3.3.5.2.3 The prescription and proscription of statistics need not be followed as strict rules

Even given the 'correct' classification of a measure as belonging to a particular scale type, the standard prescription and proscription of particular statistical techniques need not, or should not, be followed as strict rules. This idea may be supported by several points.

Non-parametric statistics require fewer or less stringent assumptions about the nature of the distributions of data. More specifically non-parametric statistics do not require normally distributed data with equal variance. They are held to be suitable for ordinal data (but not categorical data with large numbers of tied values). The Wilcoxon signed rank test for paired differences is classified as a non-parametric statistic. However, since it uses the difference between paired observations in its calculation it will clearly be sensitive to variations in interval (Briand, El Emam et al. 1996). This means that some statistics, which are considered as non-parametric, are, strictly speaking, not

suitable for ordinal data. Such interval dependent non-parametric statistics are commonly used for ordinal data and their results readily accepted. Given this acceptance, what is so unacceptable in using other interval dependent statistics for ordinal data simply because they have been classified as parametric?

Those non-parametric statistics that are interval dependent are classified as non-parametric because they are robust to high levels of non-linearity. This means that although not strictly in accordance with the allowable transformations for ordinal measures we still have a high level of confidence that the results they produce are meaningful and acceptably accurate. It is this level of confidence that is significant. If we can be confident about the results of very robust statistics on highly non-linear data might we not have the same level of confidence of less robust statistics on slightly non-linear data? The fundamental point in the selection of statistical techniques should be their robustness to the level of non-linearity in the data with which we intend to use them.

This type of careful and considered use of parametric statistics to analyse non-interval data benefits from the increased power of parametric over non-parametric statistics. The result of this increased power is an increased likelihood of finding significant results (for a given data set). Viewed from another perspective the use of parametric statistics can reduce the amount of data required (and so the data collection effort/cost) to achieve a significant result.

3.3.5.3 The requirement for complexity measures to be additive

The argument that software complexity measures should be additive has been made, initially by Zuse (1991). This argument has gained in force through its use by other researchers (Bieman and Ott 1994; Chidamber and Kemerer 1994). However, the reasons behind such a standpoint (and its validity) are questionable.

The idea seems to have been introduced because of the desire to be able to use the ratio scale. Additive measures were suggested because they are always on a ratio scale. However measures may be ratio but not additive. This means that even if we

accepted that we require complexity measures to be ratio scale, stating that they must be additive is an unnecessary restriction.

The assertion that complexity measures must be additive, and the fact that additive measures must also be ratio scale, implies that no useful ordinal or categorical data on software complexity can be gathered. This seems clearly flawed. A measure that would allow us to choose the least complex of two implementations of the same code would be a useful measure of complexity, and need only be on an ordinal scale.

The above points try to show that the assertion that complexity measures be additive is not a prerequisite of a valid or useful measure. We might go further and consider if additivity is even a desirable property of complexity measures. An additive measure is based on an extensive structure and there are a number of axioms that must be satisfied by extensive structures. Amongst these properties are weak commutativity and weak monotonicity.

If we consider a measure of code complexity the commutativity axiom would require that the result of measurement be dependent only on the program elements contained in a piece of code and not their order. This is clearly counter to the intuitive idea of code complexity. The monotonicity axiom also runs counter to the intuitive idea of code complexity. It would mean that a measure couldn't allow for additional complexity being created by the interaction of two program segments being composed.

Extensibility does not therefore appear to be a property that necessarily fits the intuitive notion of at least some aspects of complexity. It would therefore break the representation condition to force extensibility onto a property where it does not naturally belong simply to be sure of ratio scale measurement.

3.3.6 Empirical validation

Empirical validation is the process of checking whether the intuitive properties of our measures, and the theories embodied in our models, are consistent with observed results. Measure validation may be compared to software testing. In testing, even if the actual behaviour matches the expected behaviour, the system is not proved to be correct. Similarly, in validation, a study showing that a measure behaves according to our intuition or that a model corresponds the behaviour of the system does not once-and-for-all 'validate' that measure or model. Empirical validation is an ongoing process of collecting observation data, which may corroborate or challenge a measure or model.

3.3.6.1 Techniques for empirical validation

There are three main types of empirical investigation used within software engineering research. These are experiment, case study and survey (Pfleeger 1994). An experiment is a rigorous investigation into a phenomenon done under carefully controlled conditions. Case studies are investigations done on real situations with little or no control over variables (but control over data collection). Surveys are generally retrospective studies, and as such, there is no control over the situation being investigated. An in-depth description of empirical research is beyond the scope of this thesis. Further information on the use of empirical methods within software engineering can be found in the following sources (Wohlin, Runeson et al. 2000; Fenton and Pfleeger 1996; Shepperd 1995).

Empirical validation within measurement is most often carried-out using formal experiments. The structure of such an experiment and the nature of the hypothesis under investigation will depend on the nature of the measure or model being validated.

If a subjective measure is being evaluated then it is important to ensure a high inter-rater agreement. An example of this type of validation has been published by El Emam (2001; 1999). This study used the Kappa coefficient (Siegel and Castellan 1988) to gauge the levels of inter-rater agreement in subjective scores given during

software process assessments.

Indirect measures may be validated by comparing the output from the model with a direct measure of the same attribute. If a direct measure is not available an attempt could be made to validate the model itself. As has already been stated, the model can be thought of as representing a theory and this theory can be tested by experiment.

For a direct measure it may be possible to experimentally verify the empirical relational system for the attribute being measured. Many of the potential problems with measurement theoretic validation originate from the intuitive basis for the empirical relations. Kitchenham, Pfleeger et al. (1995) recommend using measures of inter-rater agreement to corroborate that the intuitive basis for a measure is at least a shared intuition.

Models capturing the use of measures can also be empirically validated. The wide range of possible measurement uses means that little can be said in general about how empirical validation should be done on such models. The validation technique will be dependent on the type of measurement use.

3.3.6.2 Combining empirical studies

As has already been mentioned, empirical validation is an ongoing activity. Each new piece of corroborating evidence helps to increase confidence in the measure or model. A new piece of evidence usually takes the form of a separate empirical study. This may either be a direct replication of a previous study, or separately designed study with similar goals. If the evidence of the validity of a particular measure or model is contained in a number of separate studies, it is essential to be able to combine these results. This is achieved through meta-analysis.

Recently, consideration has been given to the application of meta-analysis in empirical software engineering (Pickard, Kitchenham et al. 1998; Miller 1999). Initial results in using meta-analysis in software engineering have been poor (Miller 1999). It has been suggested that results may be improved to some extent by making changes to

current experimental and reporting practices. Miller (1999) makes several suggestions for such changes, including the following points:

- The inclusion of effect size estimates in experimental results as well as significance levels.
- Making the raw data from the experiment available to other researchers where possible. Alternatively, failing this, descriptive statistics or histograms of the distribution of the data should be given.
- The number of subjects included in a study is important for meta-analysis and should always be explicitly stated.
- The independence of studies is important for meta-analysis. This means that the exact replication of an experiment can be potentially dangerous. The reuse of experimental materials and procedures can lead to correlations related to the materials and not the phenomena under investigation.
- It is important that authors report non-significant results as well as significant ones. When combined with other experiments they can add important data to the meta-analysis. The meta-analysis can give misleading results if this data is not included. This is particularly important in software engineering research where data sets are typically small and even discernible trends are often not statistically significant.

3.3.6.3 Current state of empirical research

There is still little occurrence of third-party empirical validation of software measures. The reason for this may be the current attitude toward research in this area. With funding dependent on publications and the high profile initial work in an area, the 'hard-graft' follow-up work has less appeal (and less reward). Validation of someone else's work is not held in the same esteem as it is in other disciplines. It is common to see a race between rival labs in physics or biology to see if they can replicate the latest published findings. Although it is rare to see third-party validation of software measures there are some examples (Harrison, Councell et al. 1998; Basili, Briand et al. 1995).

The situation is slightly better in software engineering research in general. The ISERN network has been prominent recently in pushing for increased replication of empirical work. They are currently organising an experiment on software inspections, which will be replicated across several institutions (Wohlin 2001). ISERN have also published many technical reports including some describing replication studies (Land, Jeffery et al. 1997; Visaggio 1997; Briand, Langley et al. 1999). In a recent technical report published by ISERN, Briand, Arisholm et al. (1999) suggest that a 'replication package' should be made available to the wider community containing details of any experiment.

3.3.7 Measurement types and appropriate validation techniques

In section 3.2.3 a number of attributes of measures were classified. These attributes are measure type (direct/indirect), attribute type (internal/external), entity type (product/process/resource) and collection type (objective/subjective). To consider what validation technique(s) are applicable, all possible combinations of these attribute types were considered. However, after examining these combinations it was felt that the attribute type and entity type had no effect on the outcome.

Measure type	Theoretical validation	Empirical validation
direct objective	Measurement theoretic validation of the measure itself.	Validate the empirical relational system.
direct subjective	Axiomatic confirmation of the subjective scale.	Assess inter-rater agreement.
indirect objective	-Confirm the theory on which the attribute model is based.	- If possible, validate against external direct measure - Empirically validate attribute relationship model.
indirect subjective	-Axiomatic confirmation of the subjective scale. -Confirm the theory on which the attribute model is based.	-Assess inter-rater agreement. -Empirically validate attribute relationship model.

Table 3-1 Validation techniques for different classes of measurement

Table 3-1 attempts to summarise the relationships between validation techniques, measure type and collection type. The combination of attributes is shown in the left-hand column of the table. The next two columns show the suggested theoretical and empirical validation techniques for a measure with that combination of attributes.

It should be understood that these are only suggestions for appropriate validation techniques. There is no implication that following the above prescriptions will necessarily lead to valid measurement, i.e., there is no implication that these suggestions form a complete validation system. Neither is it suggested that other forms of validation might not be successfully employed.

3.4 Measurement processes

A useful means of structuring a discussion of software measurement is to look at the processes of measurement. When looking at the stages which measures pass through it becomes apparent that the process of measure development is different from the process of measure use. These two processes are therefore reported separately (although there is some overlap).

The measure development process describes the conception, definition and initial validation of a measure. This process considers development across the entire community (mostly the research community). The measure use process begins with a perceived measurement need within a particular development setting. It then proceeds through measurement selection, collection and analysis. The end of this process for a measure is when the measure ceases to be collected within that particular setting (for whatever reason).

3.4.1 The measure development process

This section deals with the measure development process. The stages of development of a new measure will differ for direct and indirect measures. The first subsection will describe these stages for a direct measure. In the following subsection a description is given of the stages in the indirect measure process that differ from those in the direct measure process.

3.4.1.1 Direct measures

The process stages for a direct measure are summarised in the schematic shown in figure 3-3.

3.4.1.1.1 Identifying a measurement need

New measures are (or should be) proposed by researchers when they perceive a measurement need which is not met by existing measures, or believe that a better measure can be devised to replace a problematic existing measure. New measures are often proposed for attributes already covered by other measures without any discussion of problems of the existing measures or the additional benefits of adopting the new measure. For example, Lorenz and Kidd (1994) define measures similar to (or identical to) existing measures without any reference to these previous measures.

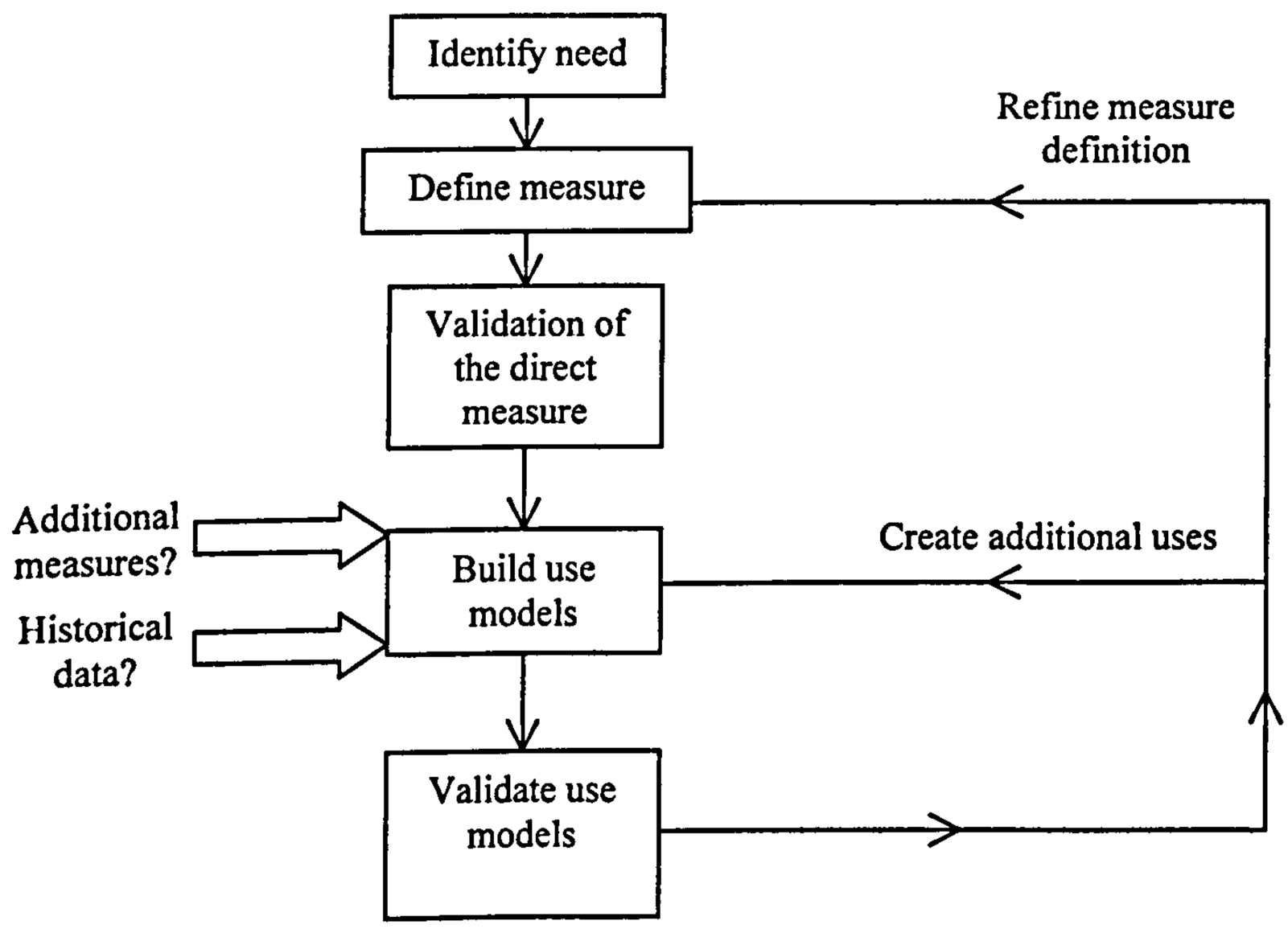


Figure 3-3 Measure development process for direct measures

3.4.1.1.2 Defining the measure

Once an author has identified a measurement need, they must formulate a definition for their new measure. Here we meet a problem in software measurement; there is no generally agreed way of defining measures. Authors vary in the way they describe their measures and the information they provide about measures. For example,

Lorenz and Kidd (1994) define their measures in what might be termed a pragmatic manner. They concentrate on the utility of the measures, examing how the measurement results should be used, looking at factors that may impact the results and giving advice on how to remedy problems highlighted by the measures. They give only simple textual descriptions of the collection rules and no theoretical validation. Chidamber and Kemerer (1991) define their measures in a more mathematical manner (mainly based on set theory). They used Bunge's ontology (Bunge 1979; Bunge 1977) and Weyuker's axioms (Weyuker 1988) for theoretical validation.

Measures have occasionally been suggested without any clear statement of basic issues such as the entity and attribute being measured. For instance, with function points (Albrecht 1979; IFPUG 1992), it is unclear what attribute of what entity is being to measured (Shepperd 1995). Some practitioners seem to use function points to measure the functionality in a specification and others to measure effort in implementation. How is the community to consider the validity or utility of a measure without even being told what is being measured?

Measurements should only be taken to help in achieving or monitoring predefined goals (this will be discussed in section 3.4.2.1). This suggests that the author should state the use(s) they envisage for their measures when they are first suggested. The measures may be put to other uses later, but it seems senseless to propose a measure without any idea of how it could be used.

There is a problem with the lack of definition of counting-rules for measures (Churcher and Shepperd 1995a; Churcher and Shepperd 1995b). Measures are often described at a conceptual level without detailed rules for how they are to be applied in practice. It is not until we come to use them in the context of a particular design notation or programming language that we find there may be a number of interpretations of the conceptual description. These differing interpretations can produce wildly differing measurement results (Churcher and Shepperd 1995a). This problem is accentuated by the variations in the constructs and syntax between the various programming languages and design notations.

Two main approaches have been suggested in order to deal with the counting-rule problem. Churcher and Shepperd (1995b) suggest a single standardised notation for describing common features of object-oriented programming languages. This has the advantage that a single description can be made when the measure is first proposed. However, for this type of description to be complete requires a notation that covers a superset of the constructs in all of the notations and languages to which the measure may be applied. This would require a large, complex notation. If new constructs were created, in new or updated languages, then the measure description notation would need to be expanded to incorporate these changes. The descriptions of existing measures may also need to be changed to allow for these changes to the measure notation. A second approach is to describe how each measure is mapped into the syntax of each language of interest. This is the approach taken by the MOOD project team (e.g., (Abreu, Goulao et al. 1995; Abreu and Melo 1996)). One possible disadvantage of this scheme is that anyone wishing to apply the measures to a language for which the original author did not supply a mapping is essentially back to square one. They would have to interpret as best they could the conceptual level description of the measure and produce a mapping for the new language that they feel best captures the underlying ideas.

3.4.1.1.3 Validation of the direct measure

A direct measure will need to be validated to ensure that its results correctly represent the attribute being measured. A number of techniques for measurement validation were discussed in section 3.3. Although any of these techniques can be applied at this stage, the author would consider a validation using classical measurement theory to be the most compelling.

If a measurement theoretic validation is performed on a measure, it is necessary to show that intuitive properties of the attribute being measured are reflected in the measure. This is the representation condition. If those initial intuitions are 'incorrect' or they are incorrectly characterised in the empirical relational system, then however mathematical and systematic the validation after that point, the measure will be

invalid.

For direct measures, we may corroborate our intuition about the empirical relational system through experiments assessing inter-rater agreement. Statistical techniques can be used to assess the how well rankings produced by a set of human judges compare to that produced by the measure, i.e., Kendall's rank-order correlation between several judges and a criterion ranking T_C (Siegel and Castellan 1988). If the inter-rater agreement is high then there is grounds for confidence that there is general agreement on the behaviour of our attribute. If there is also a high correlation with the measurement results then we also have evidence for the validity of the measure.

3.4.1.1.4 Building use models

Once the measure itself has been validated, consideration can then be given to the potential ways of applying the measure. Kitchenham's list of models discussed earlier includes only one that would be regarded as an end use of measurement, prediction (indirect measurement models use other measures but are only created for some other end purpose). To model other types of measurement use requires a more complete set of the types of uses to which measures are put. Whitmire (1997) suggests a list of five different types of measurement use:

1. assessment,
2. comparison,
3. estimation,
4. prediction,
5. investigation.

Assessment is the evaluation of a single software entity without reference to measurement values from any other entity. This means that the predetermined values or thresholds for the measurement values are used. This type of use would fit the description of a model given earlier. One or more measures as inputs with a simple pass/fail output.

Comparison is also used for evaluation, but rather than using a predefined threshold it

compares the measured values of two or more similar entities. The purpose here is to decide which is the best rather than whether any one entity is of a sufficient standard. Again this can be modelled as described above, with measurements of two or more entities as input and a ranking of these entities as the output.

Whitmire makes a distinction between estimation and prediction mainly on the basis that estimation is used to determine likely resource levels and prediction is used to determine likely future values of product measures. This distinction is not significant for the form of the model used to represent them. Both of these uses fall under the prediction system model type, as described by (Kitchenham, Pfleeger et al. 1995).

'Investigation is the use of measurement data from a product or process to support or refute an hypothesis' (Whitmire 1997). An hypothesis is a theory concerning how various entities, or their attributes are interrelated. This theory can be represented as a model.

Rather than describe a classification of measurement uses, as Whitmire does, Fenton and Pfleeger (1996) give a list of particular uses . This list consists of the following uses:

1. Cost and effort estimation

   This use fits under the heading of prediction systems as described earlier.

2. Productivity measures and models

   This appears to be a subset of management by metrics (see point 9).

3. Data collection

   Although Fenton lists data collection as a use, it is not really an end in itself. It is really only a precursor to the actual measurement use.

4. Quality models and measures

   Quality modelling and measurement concerns the evaluation of entities. Uses of this type will be included in the assessment or comparison categories discussed earlier.

5. Reliability models

   Reliability is just one aspect of quality. A reliability model is simply a quality

model focusing on the reliability aspect of quality.

6. Performance evaluation and models

   Again, this appears to be a subset of management by metrics (see point 9).

7. Structural and complexity metrics

   Producing complexity measures is not really an end in itself. These measures are likely to be used for some form of quality evaluation (assessment or comparison) or for building prediction systems.

8. Capability-maturity assessment

   This seems a strange inclusion for a measurement application. Capability-maturity concerns the processes used by an organisation for software development. As such, it would be expected that CMM assessment would be based on a qualitative examination of a company's processes and not based on measurements. It is true that the way in which measures are used would be part of the assessment, but it is hard to see how measurement would be used in the CMM assessment itself.

9. Management by metrics

   Whitmire describes three different views on software measurement technical, tactical and strategic. Project management equates to Whitmire's tactical view and upper management to the strategic view. According to Whitmire's model, management is not a distinct use but a higher-level view of measurement. The difference is in the type of decisions taken based on the results rather than on the actual use to which the measures are put.

10. Evaluation of methods and tools

    This use fits under the investigation heading presented earlier.

3.4.1.1.5 Validate use models

A measure is developed or collected because of a perceived need. This need is developed as a model describing one way in which the measure will be used. It is not sufficient that the measures used as inputs to a model are valid, it is also necessary to validate the model. Validation of the model for each type of measurement use will be discussed separately in the following sections. The validation of indirect measurement models will be discussed in section 3.4.1.2.

3.4.1.1.5.1 Prediction

In a prediction system, the link between the model inputs and the attribute of interest need only be statistical. Here, the validation of the model addresses only the accuracy of the prediction and doesn't seek to prove any causal links.

3.4.1.1.5.2 Assessment and comparison

Assessment and comparison are both types of evaluation. They are concerned with ascertaining the quality of an entity, either on a fixed or a relative scale. To validate these types of use it has to be shown that the indicated quality level or ranking is correct. Quality has a multifaceted nature that is difficult to reduce to a single dimension for this assessment. Even if a single facet of quality is selected on which to base the comparison, there is a further problem. The comparison will require a quantifiable definition of quality that is independent of the measurement use model. If these conditions are in place, it is possible in theory to empirically validate an assessment or comparison model.

In practice these types of model are rarely (if ever) validated. They are mostly presented as guidelines or heuristics based on experience or intuition rather than any rigorous proofs.

3.4.1.1.5.3 Investigation

There is a distinction between investigative models and the others described in this section. The other models need to be validated before they can be used. In investigation, the sole purpose of taking measurements is to validate the model. The model is created to represent a theory and testing the theory is done by validating the model.

3.4.1.1.6 Measurement use

Measurement use can be the initial academic use of the measure, usually for empirical validation purposes, or the full industrial use of the measure. Empirical validation techniques have already been discussed. The industrial use of measurement is covered in section 3.4.2.

3.4.1.1.7 Feedback and measurement improvement

The measure definition process shown in figure 3-3 shows feedback paths for the rework of measures. However, most authors seem content to merely define their measures and perhaps do some initial validation. There is little effort to develop and improve measures. Some refinements to measurement definitions have been done by Chidamber and Kemerer (Chidamber and Kemerer 1994; Chidamber, Darcy et al. 1998) and by Abreu, Goulao et al. (1995)[9]. Refinements for measures are also suggested by third parties, usually to correct what they perceive to be problems with the measure's original formulation, e.g., a reformulation of LCOM by Henderson-Sellers (1996a).

3.4.1.2 Indirect measures

The development of indirect measures differs from that of direct measures in several ways. Firstly, since an indirect measure is derived via a model rather than being measured directly there is no direct measurement validation. The development of this attribute relationship model is the second stage in indirect measurement development.

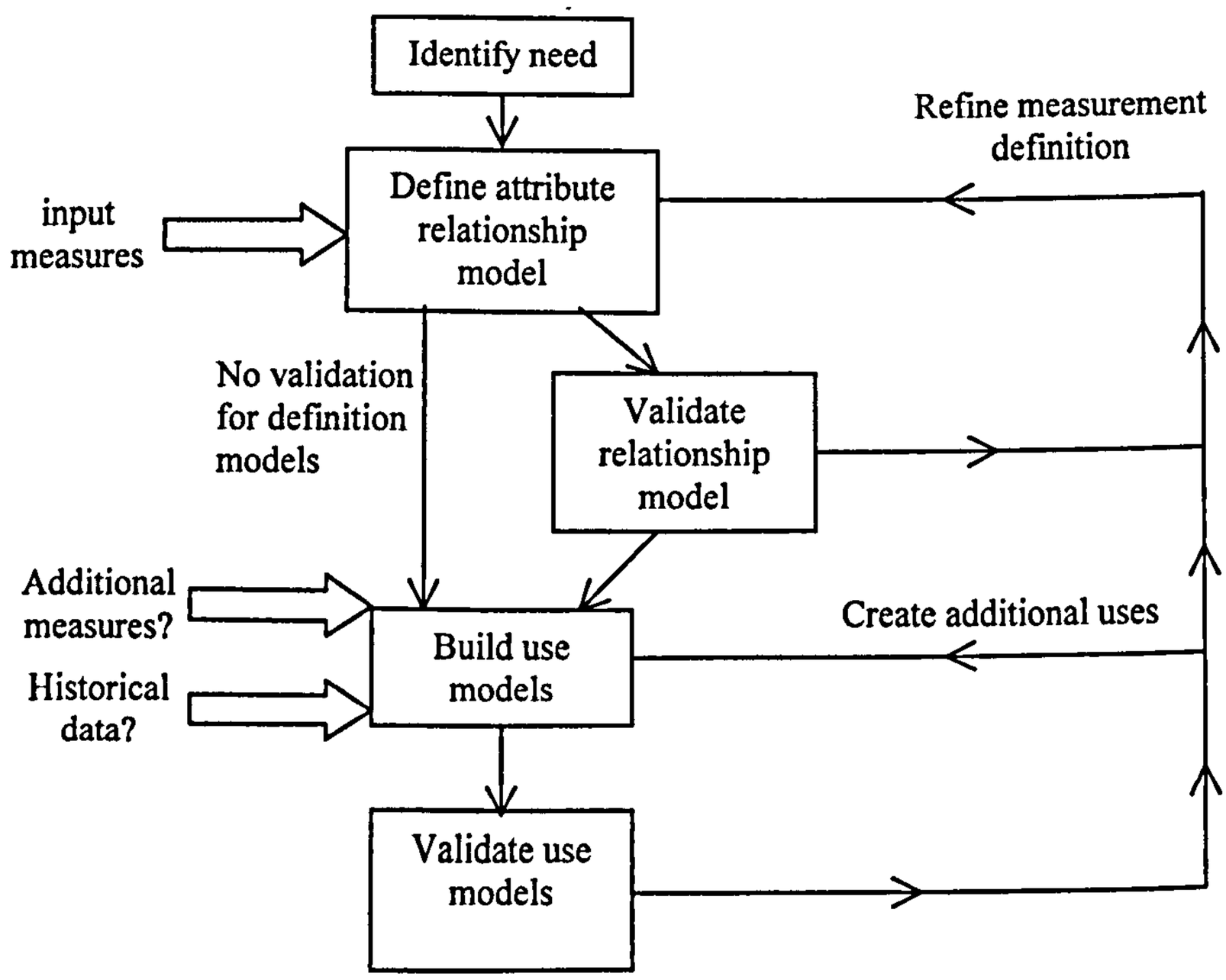


Figure 3-4 Measure development process for indirect measures

[9] Strangely, neither set of authors mention these changes explicitly in their papers. Their later papers simply show different definitions for their measures than those first published.

As mentioned earlier, attribute relationship models may be separated into indirect measures and prediction systems. Indirect measures are based around either definition models or causal models. The attribute being measured may be defined in terms of other measures or merely assessed using them. In causal models the attribute has an external definition, but there is a causal theory linking the model inputs and the attribute to be measured. In the case of definition models there is no external definition for the attribute, the model itself forms the only definition. A definition model does not require any validation, as it must be correct 'by definition' (although its input measurements must be shown to be valid). Causal models embody a theory and this theory may be empirically corroborated (though never proven). This corroboration usually takes the form of correlating the output from the model with a direct measurement the attribute. This causal model validation introduces an additional feedback path (shown in figure 3-4) to enable the refinement of the attribute relationship model if problems are found when validating this model.

As with direct measures, indirect measures are only developed if there is at least one specific use for the measure. These uses are themselves represented as models and just as with direct measures these use models also need to be validated.

3.4.2 The measure use process

This section deals with the measure use process. A schematic of this process is shown in figure 3-5. This process shows how measures are used in practice as opposed to how measures are initially developed. This process shows how measures are adopted for use in a measurement programme, how the results for the measure are collected and used, and why the measures may subsequently be dropped from the programme.

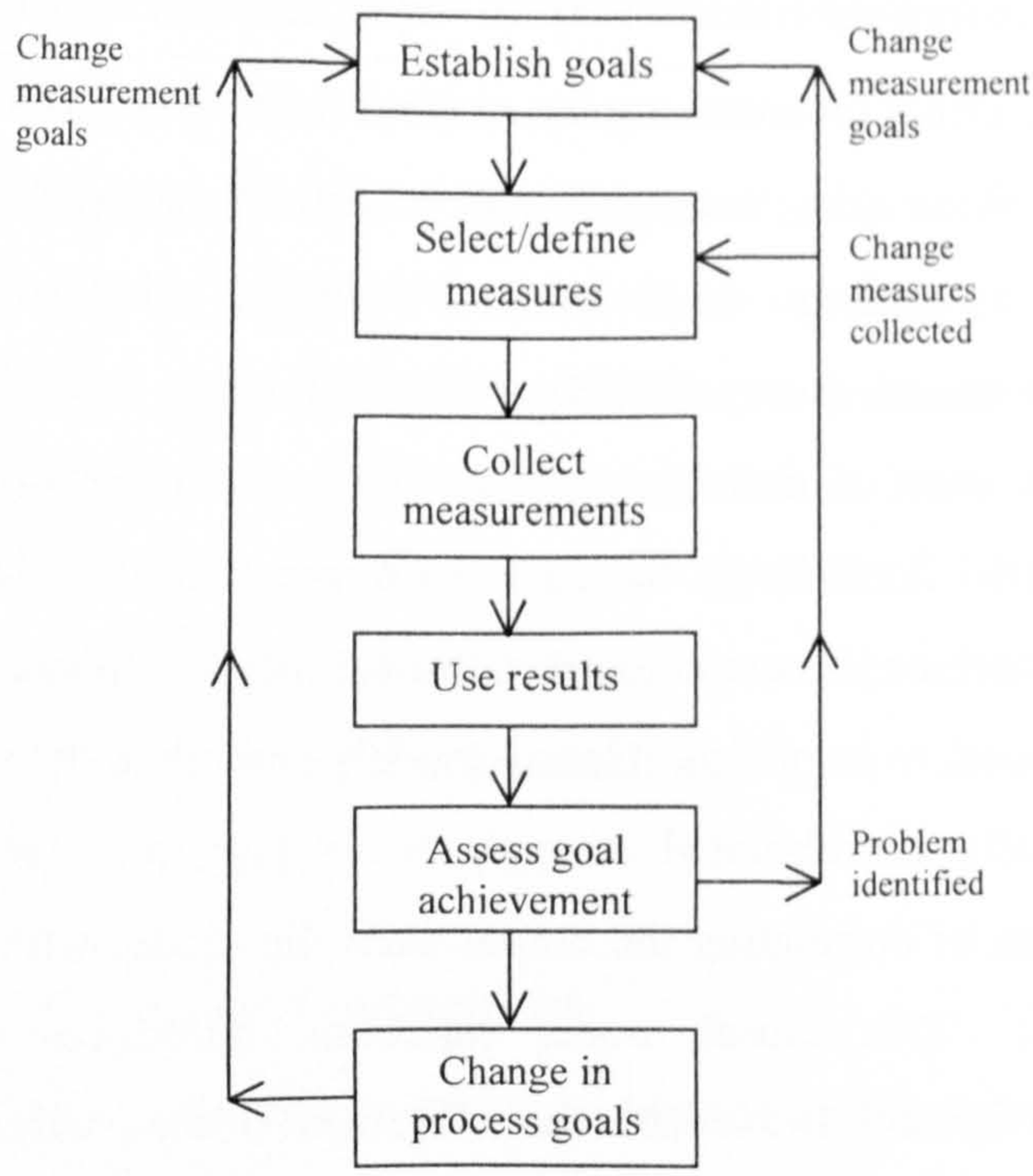


Figure 3-5 Measure use process

3.4.2.1 Establish measurement goals

3.4.2.1.1 Identify a list of possible measurement goals

Measurement sets are often derived from high level business or project goals. These goals may come from a number of different individuals or interest groups, which may be considered as roles in the software development process. The first step in identifying the set of measurement goals to be used is to identify the different roles within the project and their separate sets of goals. These goal sub-sets are then collected to form a single set of possible project goals.

One method for establishing these roles and their associated goals has been suggested as part of the Parallax MOOD project (Miller, Woof et al. 1996b). This is done within the context of a generic business structure (Miller, Woof et al. 1996a) to identify the roles as well as their responsibilities and relationships.

The GQM approach (Basili, Caldiera et al. 1994) develops goals based on three main sources of data. The first is the policy and the strategy of the organisation. The

second is a set of descriptions for the processes and products of the organisation. The third is a model of the organisation, which gives the possible viewpoints of any goal.

3.4.2.1.2 Determine the actual list of measurement goals

The collected set of possible goals may contain mutually exclusive goals, or at least goals that trade-off against each other. To help resolve these problems the importance of any goals involved in a clash must be considered by any group (role) in whose goal subset they occur. Using this information either, all but one of a set of clashing goals must be removed from the final goal set, or, an approach to arbitrating any trade-offs should be agreed.

Goals may also be removed from the list at this stage if it can be seen that the goals may not be defined in measurable terms, or if the only conceivable measurable definitions would be impractical to collect or verify.

3.4.2.1.3 Define and quantify each goal

In the GQM approach (Basili, Caldiera et al. 1994), goals are described in terms of three 'co-ordinates' and a purpose. These co-ordinates are the viewpoint, object and issue. Viewpoint is the same the role discussed above. Object is the process, product or resource being considered. Issue is the aspect of the object that is of interest, e.g., its quality or timeliness. The purpose of the goal describes the desired change to the current situation, e.g., *improve* the quality or *increase* timeliness.

There is little point in setting a goal unless it is possible, at some stage of the development, to verify that the goal has been attained. This requires that any goal be defined in measurable terms. A method similar to Gilb's quality specifications (Gilb 1988) may be employed to produce these definitions. Table 3-2 outlines the items that should be included in the definition of each goal.

Item	Description
Goal	Description of the goal
Object	The entity on which the goal is defined
Attribute	The aspect of the object to which the goal applies
Perspective	The role(s) from which the goal is derived
Test	How the achievement of the goal will be assessed
Scale	The scale-type and unit used in the test
Trade-offs	An indication of trade-offs with other goals. Indicating which goals, the nature of the trade-off, and its relative magnitude.

Table 3-2 Goal definition framework

3.4.2.2 Selecting measures

3.4.2.2.1 Deciding what to measure

Measurement should always be done with a clear purpose. This purpose should guide the decision of what to measure. The software quality methods discussed in the previous chapter provide a means of moving from the goals of the measurement to the decision of what is to be measured. Sometimes these goals are explicit (as in GQM (Basili, Caldiera et al. 1994; Rombach and Basili 1990)) and sometimes they are implied (FCM (McCall, Richards et al. 1977) has an implicit goal of assessing quality). Whatever method is used, particular measures must be selected to help assess (or help to bring about) the achievement of the goals.

Deciding what to measure is not necessarily the same as deciding what particular metrics to use. Deciding to measure class coupling could lead to the use of any one (or more) of a number of available coupling metrics. Deciding what to measure is about selecting which attributes of which entities we want to quantify (or classify).

3.4.2.2.2 Selecting particular measures

When looking to measure a particular attribute, the first question to address is whether there is a suitable existing metric. It may be that no suggestions of metrics for this attribute have been made. Alternatively, there may be several metrics to choose

between. If there are no existing measures (or perhaps if existing ones are too difficult to collect in a particular environment), a new measure must be defined.

If a number of metrics exist that purport to measure the attribute of interest, how should the metric(s) to collect be chosen? The first point to consider is the degree to which a metric's definition suits the purpose of the measurement. Generally, in software, there are no agreed definitions of attributes. Different measures tend to imply subtly different interpretations of a shared 'fuzzy' definition. This means that some metrics will capture a representation of the attribute that is more suitable for the given measurement goal.

Another consideration in selecting a measure is the type and degree of validation the measure has received. Measures are often suggested with more than one use. Extensive validation as a size measure is of little relevance if the measure is to be used to predict faults. As has been previously discussed, validation can take many forms. The mostly widely accepted form of theoretical internal validation is measurement theoretic. The exclusive use of other forms of theoretical validation might be regarded as reducing the confidence in a measures internal validity. When assessing the extent and quality of empirical validation, there are a number of factors to consider. These include the hypothesis under investigation; the artificiality of the data used; and the validity of the statistical analysis. Only empirical validation in relation to the measures intended use should be considered.

There is a trade-off between the reuse of existing measures and the definition of new measures. Existing measures may not exactly fit with the measurement goal. On the other hand, a new measure, even if defined with the specific goal in mind, is untried and requires validation.

3.4.2.3 Collecting measurement results

Data collection should be simple and unobtrusive. It should also ensure that the data collected is accurate and replicatble. This means that where possible measurement should be automated. Manual collection should have defined procedures for collection

along with centralised validation to ensure consistency. Those asked to collect data should be given proper training to ensure they understand the procedures to be followed and the rationale for data collection. Care should be taken to ensure that no unnecessary data is collected. This will include ensuring data is collected at the correct level of granularity. Where applicable, the data collected should be recorded with time-stamps and/or version numbers of the entities being measured. To enable data to be stored and retrieved simply, it should be entered into a structured database as it is collected. This enables subsets of data to be retrieved easily for analysis.

3.4.2.4 Analysing measurement results

Measurement analysis is the process of turning the raw measurement data into useful information. This includes the calculation of indirect measures or looking for patterns and trends. Considering what analysis to perform on the raw data should not be left until after the data has been collected. The analysis that will be performed on the data should already have been decided when planning the measurement programme.

3.4.2.5 Applying measurement results

Once the data has been analysed, actions need to be taken based on the results of the analysis. If future activities will remain unchanged whatever the results of measurement, then the measurement will have been a wasteful and pointless effort. Measurement results may be used for a number of purposes. Many uses of measurement directly affect process control, such as quality and productivity assessment or effort estimation. Measurement may also be used in the conduct of investigation and experimentation without any direct connection to any software process.

3.4.2.6 Assessing goal achievement

The selection of measures to collect is often based on project goals. It is an essential part of a measurement programme to assess whether these goals have been achieved. This assessment may suggest improvements to the measurement programme or show the requirement for changes to the goals themselves.

3.4.2.7 Changing process goals

As mentioned, an assessment of whether process goals have been achieved may show the need for a change in these goals. Changes to process goals may also come from many other sources, such as changes in priorities within a project or between projects. Any change in process goals, coming from whatever source, will necessitate re-planning any existing measurement programme. This in turn may result in a particular measure no longer being collected. Measures may also be dropped if new evidence shows the measures not to be valid, or if more suitable new measures become available.

3.4.3 Comparing measurement processes and metrics programmes

The standard way of considering measurement use in organisations is through metrics programmes. A metrics programme is the introduction and systematic use of software measures by a group or organisation. This includes the use of measures for project control and quality management. The programme encompasses the full range of measurement related activity from goal setting and metric selection to the collection, analysis and feedback of measurement results. Metrics programmes are often associated with quality frameworks such as CMM (Henderson-Sellers 1995; Pfleeger 1995).

Some of the standard measurement texts cover the anatomy of a metrics programme (Fenton and Pfleeger 1996; Henderson-Sellers 1996a). There are also papers covering general advice for metrics programmes (Henderson-Sellers 1995; Grady 1994; Grady 1992) and papers outlining empirical studies into metrics programme introduction (Pfleeger 1993); (Hall and Fenton 1994; Hall 1996; Hall and Fenton 1997). There has even been a framework suggested for evaluating metrics programmes (Jeffery and Berry 1993).

The measure definition process is a plan for the ideal development of a measure prior to use within a metrics programme. It is intended to further the debate on how measures should be defined, development and documented. The measure definition process is likely to be mainly of interest to academic audiences, although industrial

metrics programmes may also include the definition of new measures.

A measurement programme involves the entire set of measures being considered, collected and utilised in a project or organisation. This wide viewpoint is necessary for the success of the programme. The processes presented above are in no way an attempt to replace a metrics programme. In essence the measure use process is an aspect of a metrics programme. It is a representation of the wider system from the viewpoint of the individual measure. This change in focus allows the path of the individual measure through the process to be more clearly seen.

3.5 Summary

This chapter presented a critical review of the literature on software measurement. A brief review of OO measures from the literature was presented and the basic concepts behind measurement were discussed. This was done to provide background and context for the following discussions. Commonly used measurement terms were also discussed and defined to clarify their use in the main sections of the chapter.

The next topic discussed was the relationship between measurement and modelling. It was noted that it is only within the context of a model or theory that a measurement has a meaning. An adapted structure for modelling was introduced (based on that of Shepperd and Ince (1993)). This structure made more explicit the relationship between the formal world elements of the model and their empirical work counterparts. The various types of models suggested for use within software measurement by Kitchenham, Pfleeger et al. (1995) were also discussed. It was felt that some models in this list were more appropriate for the physical sciences than a pragmatic discipline like software engineering.

Measurement validation was the next topic tackled. Although often treated as activities within measure development, the various forms of validation are techniques that can be applied to validate different entities during different activities. These validation techniques are each described and discussed. The most commonly cited validation technique, measurement theory, is discussed at length and the value of its

rigid application questioned. There was also found to be widespread disagreement on what is needed to validate a measure. Different authors use different validation techniques to validate the same types of entities.

Techniques and issues in empirical validation were discussed and the lack of third party empirical validation for measures noted. The third party validations that have been done are mostly empirical results for the Chidamber and Kemerer metrics (Basili, Briand et al. 1996; Harrison and Nithi 1996; Wilkie and Hylands 1998) with some results for other metric sets, e.g., the MOOD metrics (Harrison, Councell et al. 1998). A summary of suggested validation techniques for different types of measures and models was also presented.

Software measurement, as it applies to object-oriented systems and OO design in particular, was also briefly discussed. It is concluded that new measures are required to describe the novel aspects of object-orientation but that some existing measures designed for structured system may still be usefully applied to OO systems and should not be ignored.

It was proposed that the activities involved in software measurement should be divided between two different (but overlapping) processes. These are the measure development process and the measure use process. The measure development process deals with the initial definition and validation of a measure, as well as any subsequent refinement and revalidation. The measure use process deals with the adoption and use of an existing measure within an organisation's metrics programme. This separation helps to remove any confusion that may occur from the conflation of these distinct sections of measurement. It provides different focuses for the academics developing measures and the practitioners using them.

Chapter 4 The design measurement tool

This chapter discusses the development and evaluation of the OO design measurement tool (a.k.a. MOT). The structure of this chapter follows the development stages of the tool. Section 1 deals with the rationale behind developing the tool. The requirements for a design measurement tool are discussed in section 2. A description of the investigative programming done to assess the feasibility of certain design options is given in section 3. A domain analysis for the tool, the tool's design, and its implementation are covered in sections 4,5 and 6 respectively. A brief description of the operation of the tool is given in section 7. An assessment of the tool is given in section 8. This involves both the testing of the tool and an evaluation of the tool. The testing principally deals with the verification of the tool's results, to provide confidence in the measurements used in the case study chapter. Evaluating the tool is done both in respect of how well it achieved its original aims and by comparison with other tools. The chapter closes by describing planned future development of the tool (section 9) and providing a summary of the major points of the chapter (section 10).

4.1 Rationale for developing the tool

The rationale for building the tool, and the approach taken to building the tool, can only be discussed in the light of what was known when the tool development started. This includes the features available in other measurement tools at that time. A new tool had to be developed because there were no tools with the desired features when the consideration of measurement needs was first carried-out (during 1997). It was also an opportunity to develop a tool for measuring systems built using Java, since none were available when the development commenced. Although many of the tool's features were novel when first conceived, some of them have since been incorporated into other measurement tools. For instance, some CASE tools now offer the ability for users to define their own measures by traversing the data model via a published API (e.g., Rational Rose'98 and TogetherJ). The intention was to show that a measurement tool could be developed which would allow user definable measures and

user selectable measurement sets.

4.2 Preliminary tool requirements

The preliminary requirements represent the ideas that were produced at the start of the tool's development. There was no formally produced requirement specification because the tool represented an investigative development. Part of the output from this investigation would be a clearer idea of what the requirements for a software measurement tool should be. Lessons learned from developing and using the tool, and from associated theoretical work, could be used to suggest a more accurate and complete specification for future measurement tools. The remainder of this section outlines the initial ideas on measurement tool requirements. It is presented to provide background for the discussion of the design and development of the tool.

In considering the requirements for a software measurement tool, one device that may be employed is to consider what such tool would do in an ideal world. If we remove those requirements that are not theoretically possible, or which are not practically feasible, we should then have viable set of requirements.

The list below is a set of ideal measurement tool requirements taken from a presentation by the author to the OOPS special group of the British Computer Society in 1997 (Kirsopp 1997). The list was compiled taking suggesting from other literature and ideas from the author's analysis of the software measurement process (Kirsopp 2001a). Areas 4 and 5 extend somewhat beyond a simple measurement tool and into quantitative process control, but after all this is an *ideal* list. These potential requirements were used in deciding the initial functional requirements of the tool.

1 Measurement collection requirements

- 1.1 Automate the collection of objective metrics
- 1.2 Automate the calculation of predefined prediction systems
- 1.3 Allow analysis of incomplete/ in-progress designs
- 1.4 Take input directly from multiple sources, e.g., CASE tools, source

code

1.5 Must be able to output results in a form suitable for further analysis

2 Project Data Storage Requirements

2.1 Should enable new metrics to be applied retrospectively to old projects

2.2 Should enable data retrieval for the production and tuning of prediction systems (although external response variables need not be stored in the tool, it should be capable supplying data for potential dependent variables)

3 Selection/Derivation of Metrics

3.1 Should support multiple view points, i.e., designer, manager, customer

3.2 Should support per-project metric selection, by considering project goals and characteristics

3.3 Should allow the definition / modification of metrics and metric-sets as indicated by measurement assessment

4 Project Planning Requirements

4.1 The ability to model the project's process including quantitative process control

4.2 Support the inclusion of measurement activities as process activities

4.3 Should include the use of measurement results as triggers/guards for development activities

5 Project Enactment Requirements

5.1 Should automatically follow the project plan prompting for progress data and project artifacts for measurement

5.2 Should automatically check triggers and guards and make indicated activity transitions

5.3 Should allow in-project re-planning

6 Measurement Research Requirement

6.1 To be able to collect data from many sources, and use that data in an integrated manner (rather than many individual studies).

This is a wish list for a measurement tool. Many of these requirements have never been built into a measurement tool. Some of the requirements may not even be

feasible. For a single researcher to attempt to produce a tool that meets all of these requirements is entirely unrealistic. What must be decided is which of these possible requirements to concentrate on. It must also be realised that the requirements are dependent on each other. Some requirements are built on the platform laid by others so that one cannot be considered unless another is first included.

The idea is to investigate requirements, not previously covered by other measurement tools, that have no dependencies outside of the proposed tool, and that can be built within the time and resources available. The dependencies tend to limit possible requirements to the 'early stages' of the measurement process. Project enactment cannot be added unless project planning is already present and neither is of any use unless measurement collection is already in place. It was decided that the tool development should concentrate on investigating multiple source-type input (OOPL and CASE), user definable measures and user selectable measurement sets.

4.3 Investigative programming

Initially, some of the ideas for the functionality of the tool were somewhat speculative. An initial period of investigative programming was done to decide the plausibility of these ideas. Taking measures from several types of sources could be achieved by converting each to a common intermediate format first and then calculating the measures on the common format. This would allow the measures to be defined only once, and not redefined for each source type. Storing the design information and generating measures from the stored information seemed a more tractable approach than creating a parser into which user defined measures could be added. Several possible storage schemes were considered including, a relational database, an object-oriented database, and a custom made persistent object store. Of these the relational database was thought the most desirable (if it could be made to work). This was because the technology is well established and widely understood. Users of such a system could store design data on their existing database and use a language that they already knew (SQL) to describe the measures.

Investigative programming was carried-out using a simplified data model and a few well-known measures. It was found that measures could indeed be coded in SQL and used to calculate results on designs stored in a relational database. This work was sufficiently encouraging to attempt this scheme for design storage and measurement description in the full prototype.

4.4 Domain analysis

The domain analysis for an OO design measurement tool is an investigation into the area of application of the tool. This would include investigation of object-oriented design and of design measurement and its uses. The end result of this investigation is a fuller understanding of the problem domain for the developers and models of the domain that may be used as the starting point for the design of the system. This type of investigation and modelling has already been presented in the preceding chapters and there seems little point in reiterating it here. This discussion will therefore move directly to the architectural design of the tool.

4.5 Architectural design

The section on architectural design is split into two main sections, the first of these deals with the design of the tool itself. The second section deals with the design description language (DDL) used in the tool.

4.5.1 High-level design

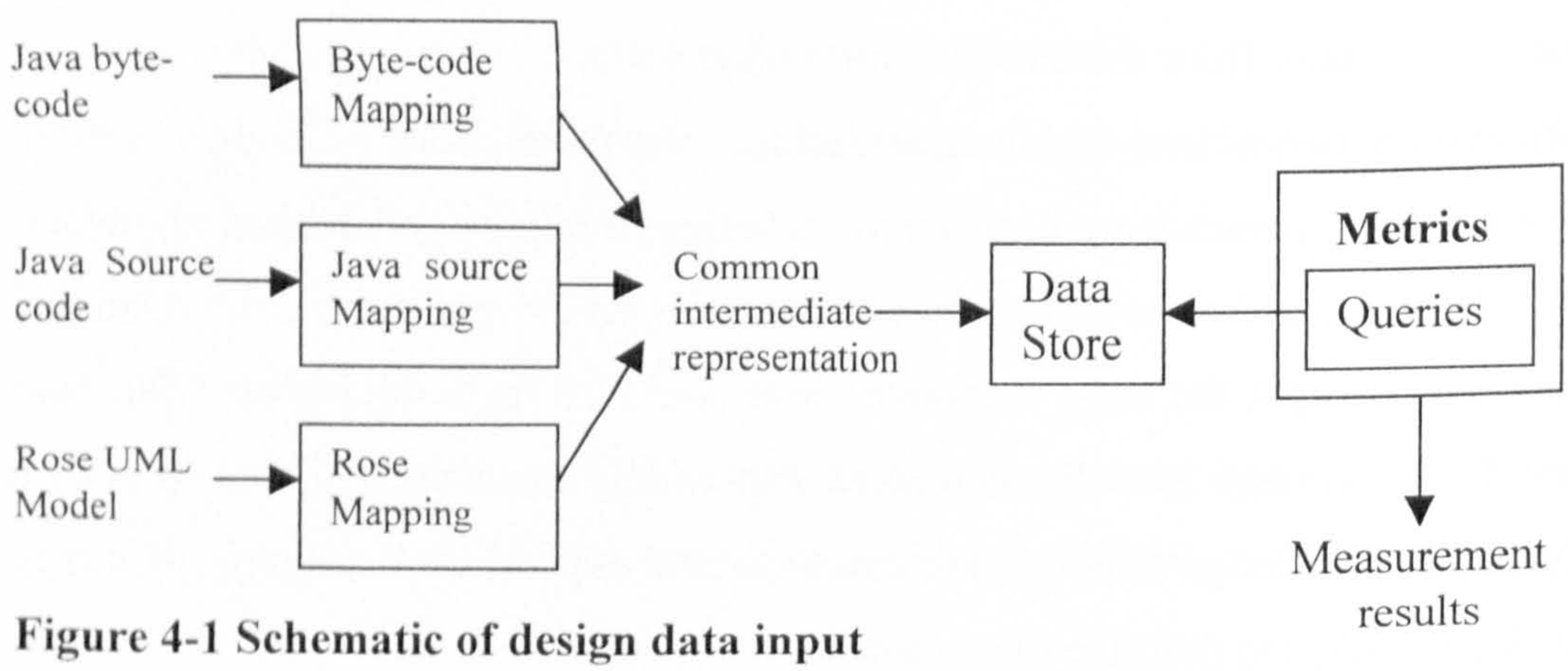


Figure 4-1 Schematic of design data input

A basic schema showing the arrangement for data input into the tool is shown in figure 4-1. The tool can take input from multiple sources of design information

including program source code and CASE tool design diagrams. Each type of input has an associated mapping that converts the design information into a common intermediate representation suitable for input into the tool's datastore.

Figure 4-2 is a UML class diagram showing the central architecture of the MOT tool (classes for peripheral functionality, support and the user interface have been omitted for clarity). The architecture can be considered in two central sections. There is a set of classes forming the core of the tool. These classes allow the definition of 'Project' objects each containing a number of 'SoftwareProducts'. The core classes also define interfaces for metric selection component classes and datastore component classes. A 'MetricUse' object encapsulates a metric and the particular set of parameters which will be used for that metric within a given metric set. This allows a parameterised metric to be defined once but used in different metric sets with different parameter values.

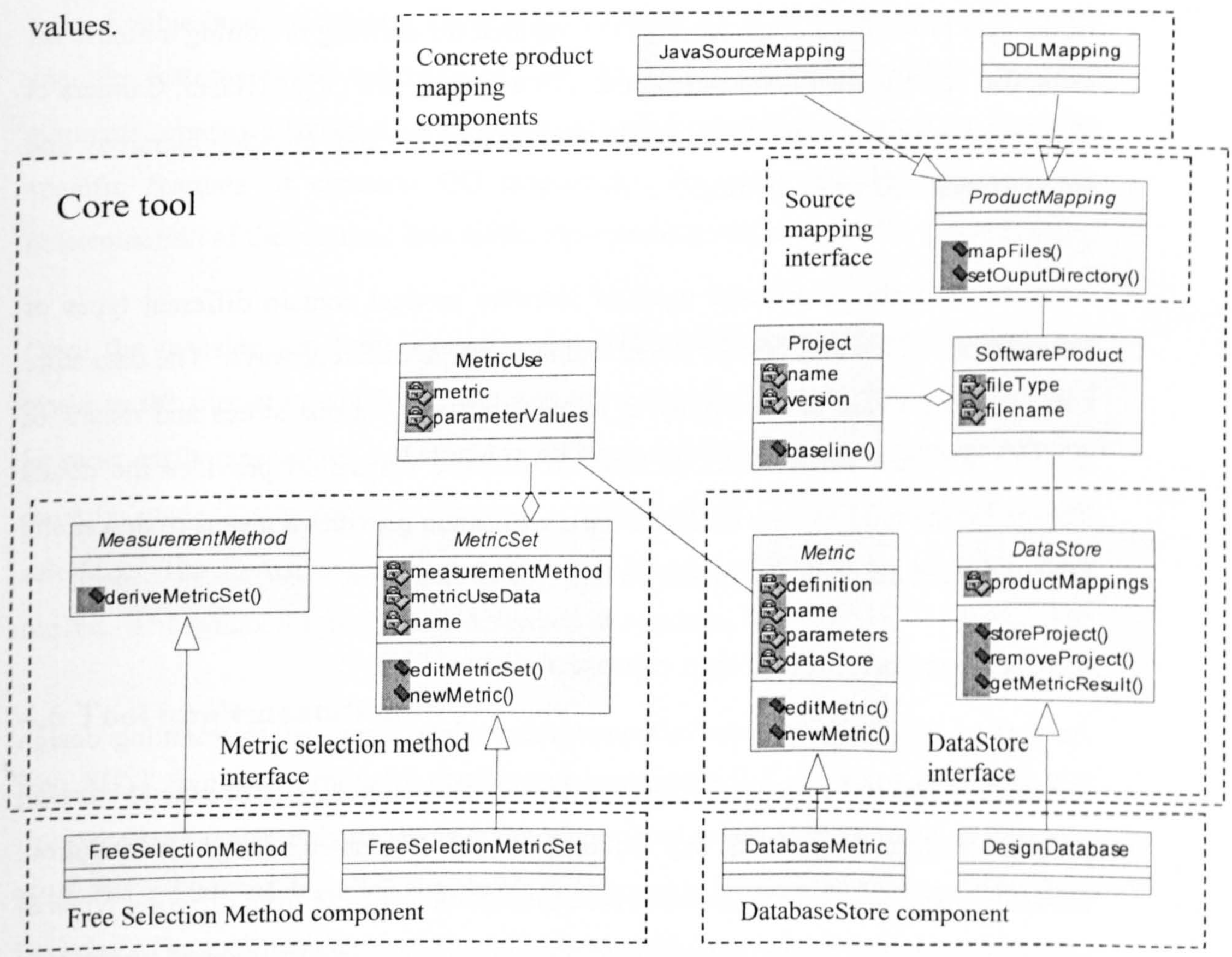


Figure 4-2 Measurement tool architecture

The second major section of the architecture is the set of pluggable components that provide concrete implementations of the component interfaces defined in the core tool. Implementing the components via these interfaces provides a 'dependency reversal' enabling new components to be added without any changes being required in the core tool.

Any 'metric selection method' component must export classes that provide concrete implementations of the MeasurementMethod class and the MetricSet class. Components may also include any necessary support and user interface classes. A MeasurementMethod realisation implements the functionality for a particular method of selecting metrics. Its associated MetricSet realisation allows the metric sets produced by any measurement method to be used in the same way by the core tool whilst also providing storage for any data required for deriving or editing a metric set using that specific measurement method. For example, in a GQM (Basili, Caldiera et al. 1994) component a metric set would need to store the goal-question hierarchy that led to the selected set of metrics.

As mentioned above, different types of software product contain different types of information, which may best be stored and analysed in different ways. The data store components provide the flexibility to implement these various stores and means of analysis within a single tool. A ProductDataStore realisation provides the actual storage for the product data whilst a Metric realisation provides a means of describing how to calculate a particular metric on that type of data store.

4.5.2 Object-oriented design representation

As part of the design process, a source-independent means of representing design information had to be designed. Ideally, this representation should be able to store all pertinent information from any source of object-oriented design information. However, there is a wide variation in the information available from differing types of information source. Design sources tend not to have such detailed method or attribute usage information as source code. Different design notations contain different

information from different types of diagrams. Different languages contain different constructs, e.g., Eiffel's constraints, C++'s friend functions and friend classes, and Java's interfaces and inner classes.

In storing information from such a potentially wide variety of sources, the representation format must either contain a superset of all the constructs from all of the sources, or some level of information loss must be considered acceptable. Since the tool is intended to collect design measurement data, the storage format was based on design rather than implementation constructs. The emerging domination of UML (Rational 2000) made this notation the obvious basis for a design description format. The UML meta-model describes the constructs and relationships available within a UML model, and this was used as the basis of the design description language (DDL). The UML notation also allows for the expansion of the meta-model via stereotypes, tagged-values and constraints. These elements allow some of the notation or language specific features to be reintroduced into the DDL. Indeed the UML 'standard elements' creates a standard set of commonly used extensions covering the language specific features of common OO languages. An extensive description of the determination of the required data model structure is given in appendix C.

Once the decision had been taken on what information to store, consideration was given to the format in which the data was to be stored. Different data sources would be most easily represented and stored in different formats. As a compromise solution must be made, it was decided compromise on simplicity, a flat-file data format was selected. The format used was a simple text-based and human readable tag-value format. The syntax for the DDL is described in appendix D.

4.6 Tool implementation

The MOT tool was originally constructed using Java 1.1. It has a graphical user interface built completely in Swing components (part of the Java Foundation Classes). With the release of Java 1.2 the Swing components became part of the core Java libraries. The tool was therefore altered to work with Java 1.2. The tool contains over

100 classes and is around 20 KLOC. To discuss the implementation down to the class-level would be unnecessarily detailed and overly long. This section is intended to give the reader a feel for the general structure of the tool and the technologies used in its construction. It is not intended to provide information at the level that would be required to maintain the system.

As described in the architectural design, the tool consists of a central core and sets of peripheral components that plug into interfaces defined in the core. The components that are to be used are specified in an initialisation file and dynamic class loading is used with these components, as and when they are needed. As well as providing the interfaces and loading-mechanisms for components, the core provides functionality that is independent of these components, i.e., functionality that is common to any combination of plug-in components.

The core handles the top-level interaction with the user. It handles events from the main tool frame and calls methods on the necessary components to perform the requested functions. The core also displays status information and error feedback to the user, as well as maintaining an activity log. Thread handling is managed from the core, as is the output of results. Results may be displayed in GUI tables, HTML reports, or output in tab-delimited format for further analysis in statistical packages or spread sheets.

There is currently only one datastore component. This component uses a relational database to store information about system designs on a schema based on the UML meta-model. The database is connected to the component using JBDC (Java DataBase Connectivity) technology. This allows any database with either a JDBC driver, or an ODBC driver (via the JDBC-ODBC bridge) to be used with the tool. The database is populated from design description language (DDL) files created by the product mapping components. The information is input into the database via a parser built using Sun's JavaCC (Java compiler compiler). Once the design data is in the database, metric results can be calculated using measures defined as parameterised SQL queries. Ad hoc investigation of the design database using SQL is also possible.

A description of the database tables and valid data values is given in appendix E.

Three product mappings are currently available. These mappings deal with, Java source code, Java byte-code and Rose design models. As has been mentioned earlier, one of the initial reasons for the development of the tool was that there was, at that time, no tool support for measuring Java systems. This is the reason why Java was chosen as the first of the source mappings. Rational Rose was chosen as the next mapping to provide proof of concept for the idea of a single tool being able to take measurements from both code and designs.

The byte-code mapping uses dynamic class loading and reflection techniques to collect design information from pre-compiled Java. It is particularly useful for third party component libraries (where the source code is unlikely to be available). This mapping could be used to perform quality assessments of such component when considering them for reuse.

A C++ mapping is desirable because of the large amount of C++ used in OO development. Initial investigations into producing a C++ mapping met with problems because of the variety of dialects available. What was a keyword in one variant could be an identifier in another and so any parser had be specific to an individual C++ compiler.

All of the mappings output design information as DDL files. The Java source code mapping is built using JavaCC and JTree to construct the parser and parse-tree. Dynamic class-loading and introspection are used in the Java byte-code mapping. The byte-code mapping produces a less complete design description than the source code mapping because there is less information available from introspection than from the source code parsing. The Rose mapping is written in Rose's own scripting language and can be installed as a tool within Rose itself. When this tool is activated, it produces a DDL output file that describes the currently open design model. Details of how the various source constructs are mapped onto the data model are given in

appendix F.

The measurement method interface allows different methods of measurement selection and derivation to become part of the tool. Currently, time has not been available to build any complex selection/derivation components. A basic measurement method has been created called the 'free selection method'. This is a 'place-holder' component, which allows the undirected selection of measures by the user.

4.7 Tool description

After discussing the analysis, design and implementation processes that have gone into producing the tool, it is probably worth spending a little time describing the finished tool itself. This section covers the main elements of the tool's functionality with the aid of screen shots showing window layout and example data.

The first screen shot (figure 4-3) shows some of the windows used in defining and editing new measures. Example data is shown for the CBO metric. The top (internal) window is the main metric window. It allows the user to specify the number of input parameters to the metric and the number of columns in its result set. An editor for writing and editing the SQL script for the measure can also be launched from this window. The right-hand lower window shows the dialogue for setting or editing the report format for the results of the measure. Actual measurement values replace the '*place-holder*' elements when the report is generated.

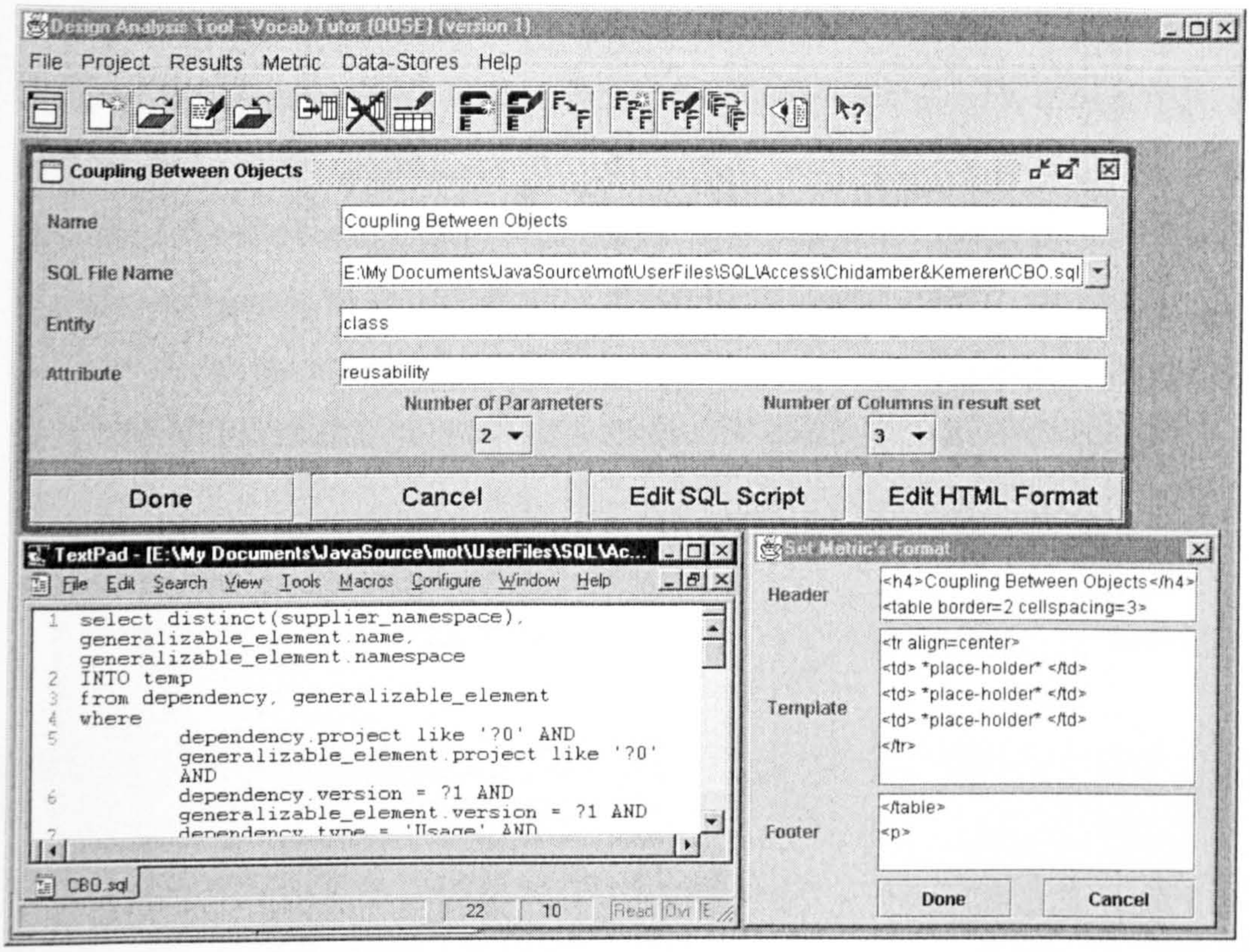


Figure 4-3 Screen shot of metric definition

Once individual measures have been defined they can be collected together into user-defined metric-sets. Figure 4-4 shows a screen shot of creating a metric-set. Example data for the Chidamber and Kemerer metric set is shown. The top inner window is the main metric-set window. From this window measures may be added to (or removed from) the metric set. This screen also allows the customisation of parameter values of the individual measures for the current metric-set. The left-hand lower window shows the setting of a parameter for the DIT metric to allow the inclusion of system classes in the depth of inheritance tree calculation. The report format of the entire metric-set can also be customised from this screen (right-hand lower window). Formatted metric results replace the '*placeholder*' elements when the report is generated.

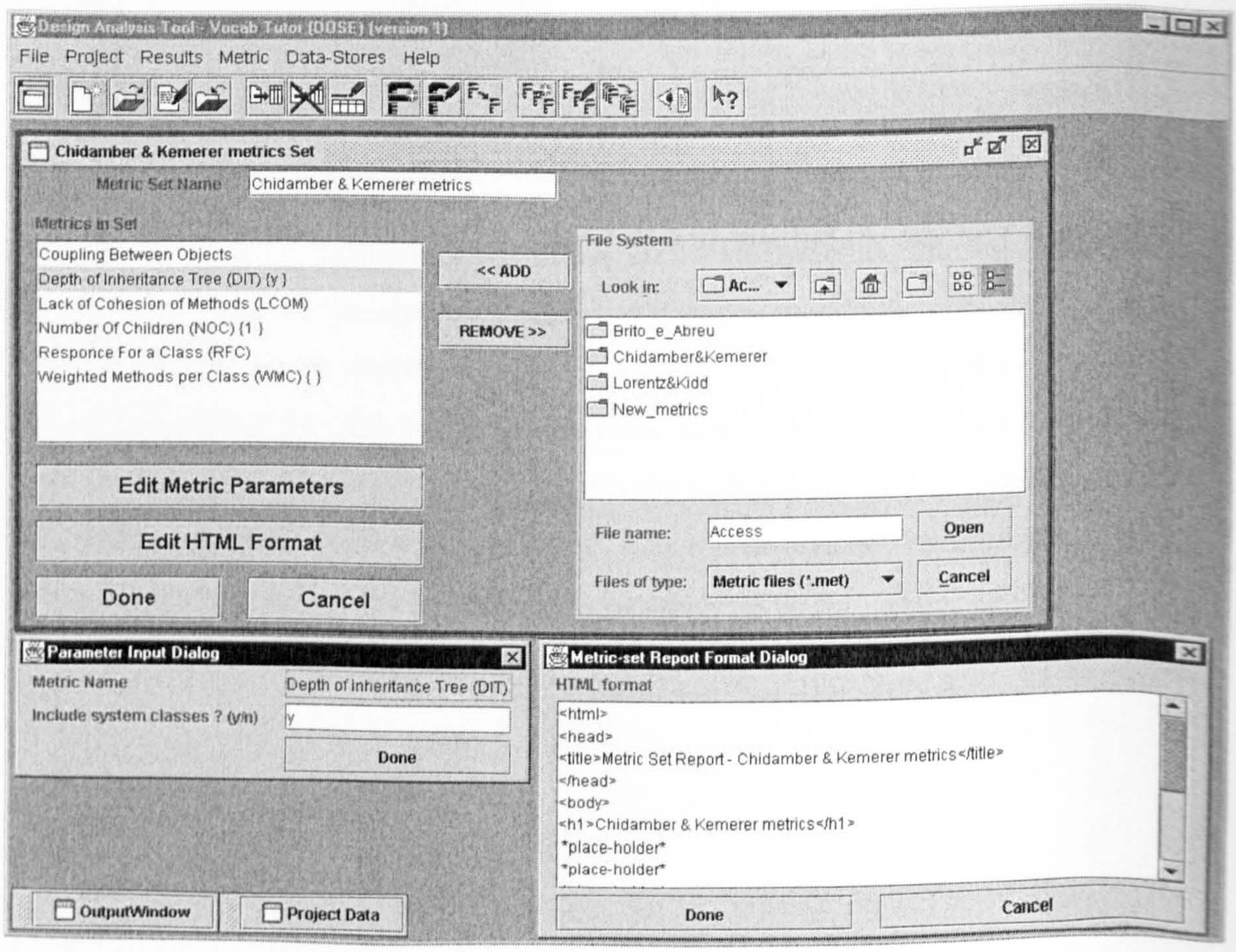


Figure 4-4 Screen shot of metric-set definition

In order to make any measurements a 'project' must be created to define the files that are to be measured and the mapping that should be used to convert the files into the internal design storage format. The screen shot below (figure 4-5) shows the project data window with example data from the 'Vocab Tutor' program used in the case study.

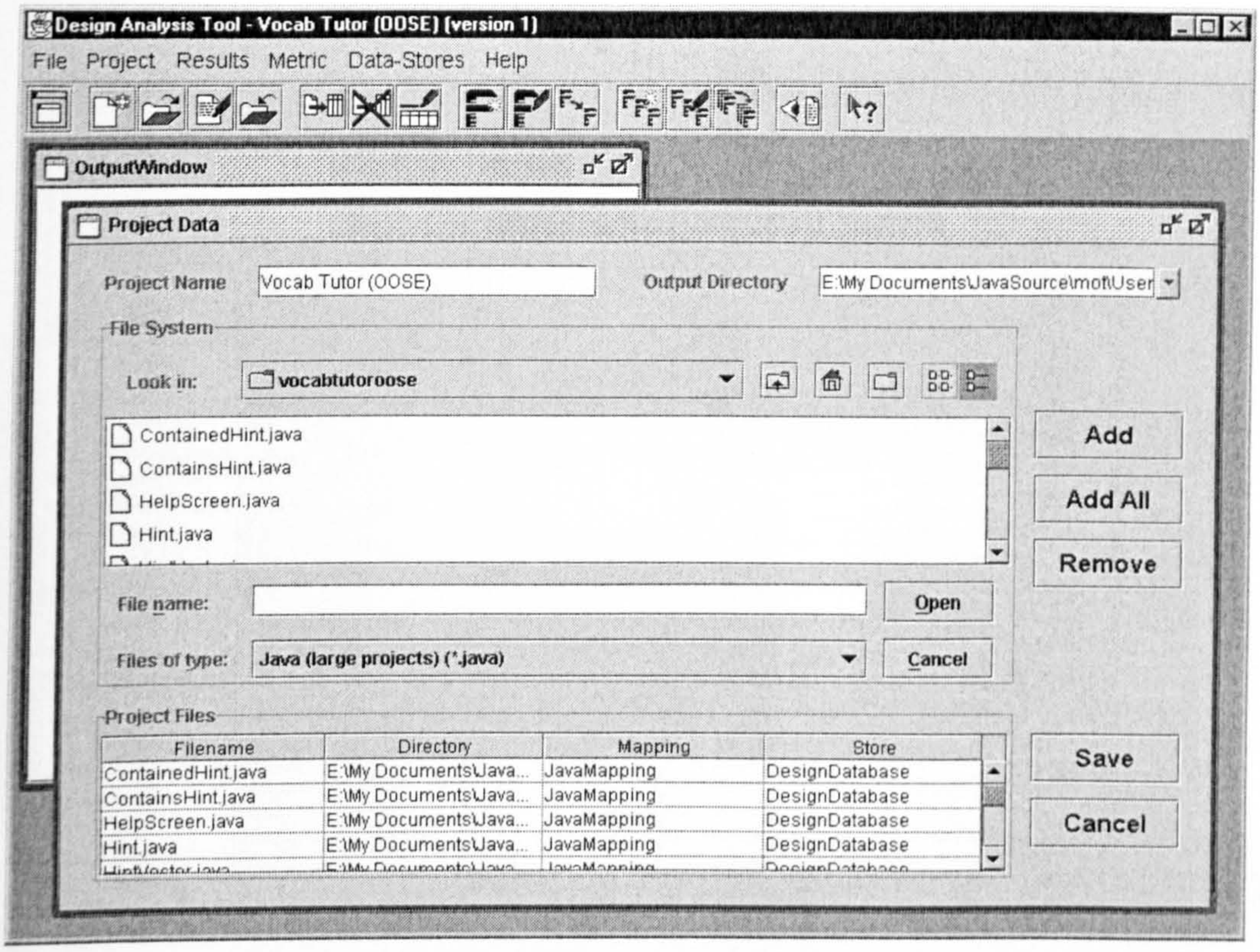
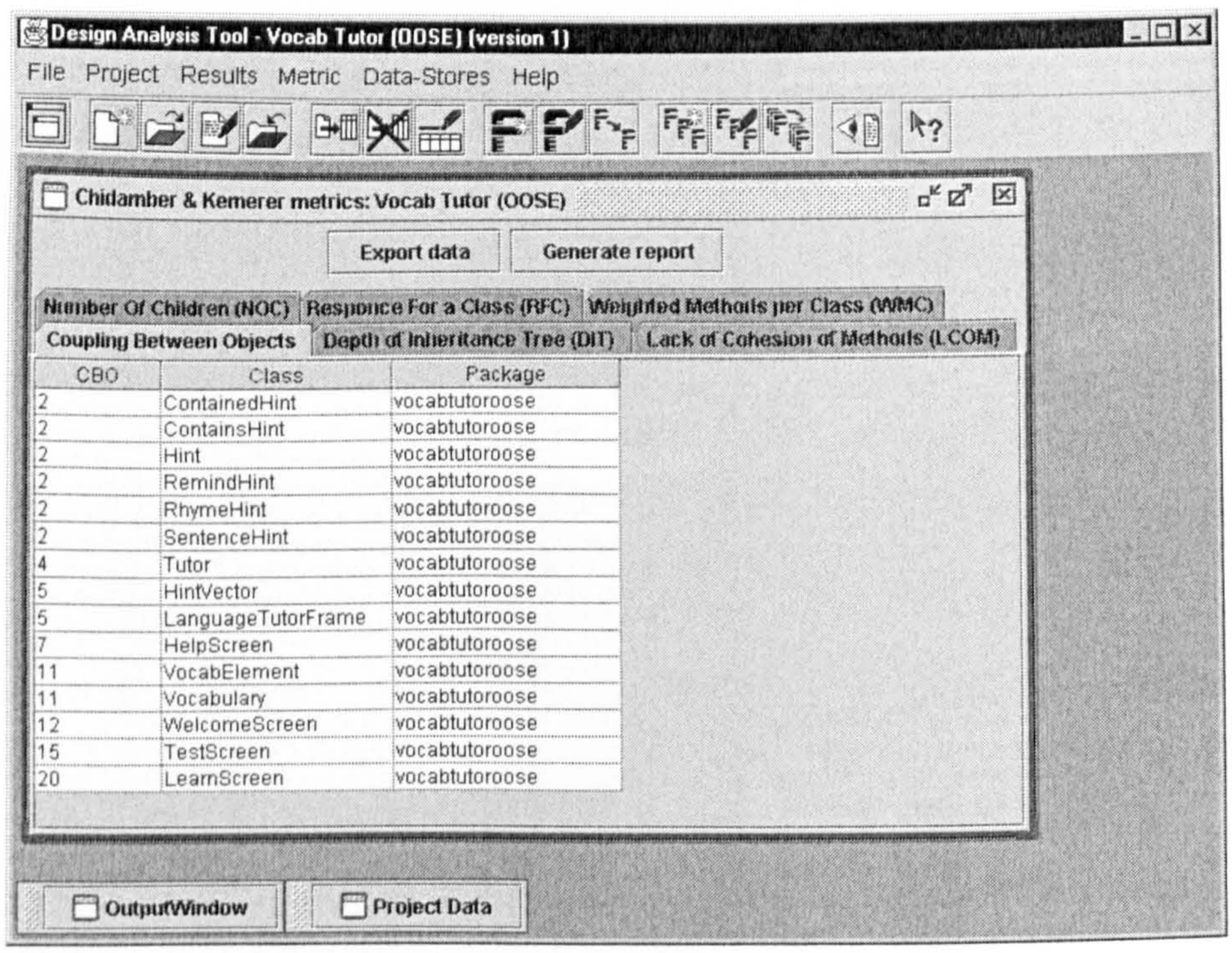


Figure 4-5 Screen shot of project set-up

When generating measurement results the user may chose to apply individual metrics or metric-sets to a particular project. Figure 4-6 shows the output window created by running the Chidamber and Kemerer metric-set on the 'Vocab Tutor' project. A tabbed set of panels is created, one tab showing the results from each metric. From this screen the user may chose to export data for further analysis in a customisable delimited text format. They may also choose to generate an HTML formatted report of the results.



CBO	Class	Package
2	ContainedHint	vocabtutoroose
2	ContainsHint	vocabtutoroose
2	Hint	vocabtutoroose
2	RemindHint	vocabtutoroose
2	RhymeHint	vocabtutoroose
2	SentenceHint	vocabtutoroose
4	Tutor	vocabtutoroose
5	HintVector	vocabtutoroose
5	LanguageTutorFrame	vocabtutoroose
7	HelpScreen	vocabtutoroose
11	VocabElement	vocabtutoroose
11	Vocabulary	vocabtutoroose
12	WelcomeScreen	vocabtutoroose
15	TestScreen	vocabtutoroose
20	LearnScreen	vocabtutoroose

Figure 4-6 Screen shot of results window

The tool comes with extensive help facilities build using the JavaHelp API. The help system has the standard contents, index and search facilities. It contains help information for the various screens and menu items. 'How to...'s for standard tasks and tutorials covering on background measurement material are also covered. For any users wishing to extend the tool with custom components, the help system also covers details of the tool's extension API.

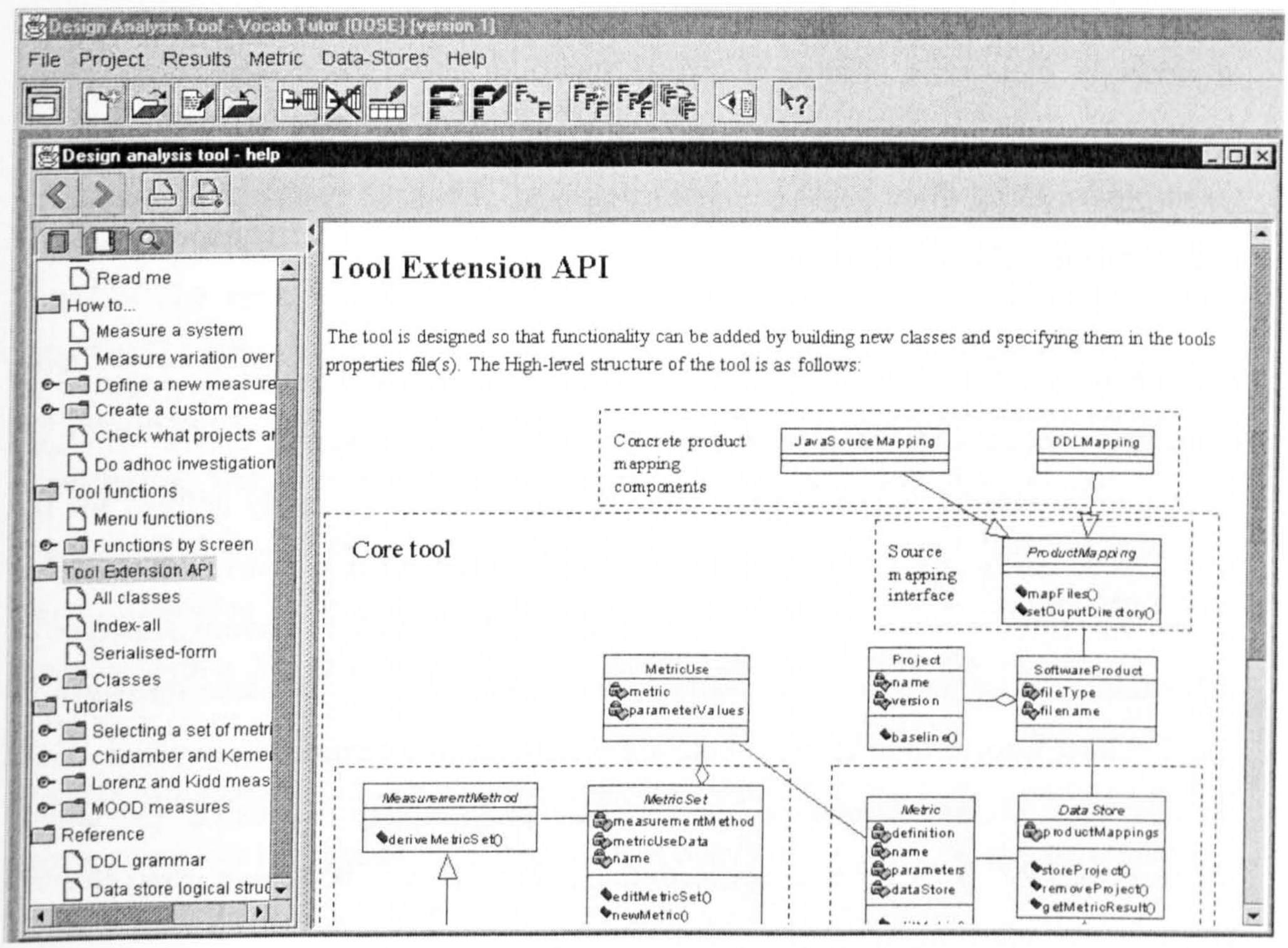


Figure 4-7 Screen shot of help system

4.8 Evaluation of the tool

The tool can be evaluated from several perspectives. Normally we would consider validation, verification and evaluation. A difficulty in trying to validate this piece of software is its exploratory nature. It is intended as a prototype tool and one of the purposes in producing it is to help decide what functionality such a tool should

provide. This means that there is no a priori specification against which to validate it. It is possible to verify the correctness of its results and to assess to tool's functionality. Evaluating the current tool can lead to suggestions for improvements in the functionality and design of future measurement tools.

4.8.1 Verification of measurement results

4.8.1.1 Possible types of verification

In principle, there are a number of different ways in which the measurement results produced by the tool could be verified. These include comparing the results against those produced by other tools, comparing against published results and comparing against results calculated by hand.

For a third party tool to be useful for results verification it must be capable of collecting measures from the same type of source (e.g. Java source code). The tool must also be able to collect at least some of the measures currently defined for the MOT tool. Although this may not seem to place too great a restriction on other measurement tools, it has proved difficult to find suitable tools to perform this type of validation. Those tools that are available free collect few of the same measures or don't collect from Java or Rose. There are tools available commercially which could do this, e.g., Metamata metrics, but these are prohibitively expensive[†] (Metamata metrics costs $995). A limited verification of the results will be shown using JMetric (Java source code) and MetricsOne (Rose models).

A comparison against published results requires that measurement results be published for freely available code or designs in either Java or Rose. No such published results have been found. The limited nature of the other verification means that the bulk of the verification must rely on comparison against hand calculated measurement results.

[†] At the time of this tool evaluation (1999) a licence for MetaMata metrics was $995.

4.8.1.2 Comparison against hand calculation

The comparison against hand calculated resulted was performed for over twenty individual measures from the sets proposed Chidamber and Kemerer (1991; 1994), Abreu et al. (Abreu and Carapuca 1994; Abreu, Goulao et al. 1995) and Lorenz and Kidd (1994; 1993). The system used for the comparison was the same system used for the case study described in chapter 5. This system consists of a single package containing 15 classes and totalling around 1500 LOC. The rationale for this choice is simple, if we verify that the results for the case study are correct, then even if bugs are found in the future, the outcomes of the case study will still be valid.

During the comparison a number of discrepancies between the two sets of results were found. The most common causes of discrepancies were mistakes in the hand calculation. Where the error was with the tool, the problems were tracked to errors in the product mapping or in the metric definition scripts. Once found these errors were corrected so that at the end of the comparison the hand calculations and the tool results corresponded perfectly. Although these tests were reasonably thorough they cannot of course be exhaustive. It is still perfectly possible that if this procedure were repeated with a different set of classes that other errors would be found.

A point worth noting in support of using measurement tools is that although it took only a couple of minutes to make the calculations using the tool, to do the same work by hand took two full working days.

4.8.1.3 Comparison against other tools

4.8.1.3.1 JMetric

The JMetric Java metrics analyser is a freely distributed measurement tool produced by Andrew Cain and Rajesh Vasa at the Swinburne University of Technology in Melbourne (Cain 1999). JMetric includes some of the Chidamber and Kemerer metrics, which are also collected by the MOT tool. Table1 shows a comparison of results produced by both tools when run on the 'second language vocabulary tutor'

program (used in the case study). The shaded cells shows differences in the results produced by the two tools.

The amount of shading on the table clearly shows substantial disagreements between the tools (27 of 60 results differed). Each metric will now be taken in turn and the differences in results considered.

The WMC results are largely in agreement, with only a single discrepancy. This is caused by a difference in the counting rules used by the two tools. In Java a class with no explicitly defined constructor methods has a constructor provided for it by the compiler. As this method exists, inasmuch as it can be explicitly called, the MOT tool includes such implicit constructors in the WMC count. The JMetric tool counts only the explicitly defined methods.

	WMC		CBO		NOC		DIT	
Class	**JM**	**MOT**	**JM**	**MOT**	**JM**	**MOT**	**JM**	**MOT**
ContainedHint	4	4	2	2	0	0	1	2
ContainsHint	4	4	2	2	0	0	1	2
HelpScreen	8	8	3	7	0	0	1	5
Hint	7	7	2	2	45	5	0	1
HintVector	5	5	4	5	0	0	1	2
LanguageTutorFrame	10	10	3	5	15	3	1	5
LearnScreen	4	4	9	20	0	0	1	6
RemindHind	4	4	2	2	0	0	1	2
RhymeHint	4	4	2	2	0	0	1	2
SentenceHint	4	4	3	2	0	0	1	2
TestScreen	4	4	10	15	0	0	1	6
Tutor	1	2	4	4	0	0	0	1
VocabElement	24	24	13	11	0	0	0	1
Vocabulary	7	7	9	11	0	0	0	1
WelcomeScreen	4	4	7	11	0	0	1	6

Table 4-1: Comparison of results from MOT and JMetric

In the CBO results the differences are far more marked. An examination of the collaborators listed by the JMetric tool showed substantial differences in counting rules. JMetric takes a far more restrictive view of collaboration than the MOT tool. It does not seem to include superclass constructor calls or explicit constructor invocations made via the 'new' construct. JMetric also includes references to basic

data types (such as int, float) in its counts. This seems strange as basic types are not classes in Java and so variables of basic types are not objects. If they are not objects, why include them in an object coupling measure?

The NOC results are somewhat odd. Both tools agree on the classes that do not have any children but differ on the number of children for the classes that do have subclasses. The MOT results are easily verified as correct from a brief examination of the source code. The JMetric results cannot be explained by any differences in counting rules or interpretation of the meaning of the measure. How can a class have more children than exist in the system? It can only be concluded that the very high values for NOC are due to an error in the JMetric program.

There is a complete absence of agreement for the DIT results. There are a number of reasons that may, to some extent, explain these differences. The MOT tool (or at least this version of the DIT metric) includes library classes in the DIT results. This is why the GUI classes, which extend existing GUI hierarchies, have high DIT values. Another counting rule consideration with Java systems is whether to include the link to the 'Object' base class in the count. In Java, any class that does not explicitly extend another class implicitly extends Object. MOT includes this link to Object in its count. Not including Object can lead to inconsistencies, since it is possible to extend Object explicitly, as well as implicitly, and as these situations are equivalent, they should produce the same result. However, the above variations in counting rules are not sufficient to explain the differences in the results between the two tools. Even if JMetric were not counting library classes, their results are not consistent. LearnScreen extends LanguageTutorFrame but both classes are given the same value for DIT.

4.8.1.3.2 MetricsOne

MetricsOne (Number-Six-Software 1999) is a commercial measurement system that operates as a plug-in tool within Rose. The demonstration version of this tool has a number of measures that can be compared with results acquired using MOT's Rose mapping. The system compared here is again the vocabulary tutor program used in the

case study. Rose's reverse engineering facility was used to produce a UML model of the system and then both tools were run on this model. There appear to have been problems with the reverse engineering in the Rose CASE tool. Many of the methods, attributes and relationships are missing from the reverse engineered UML model. This means that many of these measures are not comparable with those taken from the code. However, both sets of measurements were taken of the same Rose model (accurate or not) and so these results should at least be comparable with each other.

Class	Inheritance Depth		Children		Class Dependencies		Operation Summation		Attribute Summation	
	MO	MOT	MO	MOT	MO	MOT	MO	MOT	MO	MOT
ContainedHint	1	1	0	0	1	1	4	4	0	0
Hint	0	0	5	5	0	0	0	0	0	0
Vocabulary	0	0	0	0	0	0	0	0	1	1
VocabElement	0	0	0	0	0	0	0	0	0	0
Tutor	0	0	0	0	0	3	1	1	3	0
WelcomeScreen	1	1	0	0	2	1	0	0	0	0
LearnScreen	1	1	0	0	2	1	0	0	0	0
TestScreen	1	1	0	0	2	1	0	0	0	0
LanguageTutorFrame	0	0	3	3	0	0	0	0	0	0
SentenceHint	1	1	0	0	1	1	4	4	0	0
RhymeHint	1	1	0	0	1	1	4	4	0	0
RemindHint	1	1	0	0	1	1	4	4	0	0
HintVector	0	0	0	0	0	0	0	0	0	0
HelpScreen	0	0	0	0	0	0	0	0	0	0
ContainsHint	1	1	0	0	1	1	4	4	0	0

Table 4-2: Comparison of results between MOT and MetricsOne

The results in this comparison have far fewer discrepancies than the previous set. There is full agreement between the tools except for two of the measures, i.e., 'class dependencies' and 'Attribute summation'. The difference in the attribute summation is due to a difference in counting rules. The MetricsOne tool includes associations in its count of attributes but MOT does not.

Differences in results on the 'class dependencies' measure are not so easily explained. The author does not have access to the internal workings of the MetricsOne tool and so does not know which dependencies it is counting. Examination of MOT's set of dependencies does provide one possible explanation. The 'Tutor' class contains 3 static variables that are of types, WelcomeScreen, LearnScreen and TestScreen. In the

MetricsOne results, Tutor is shown as having no dependencies and the three classes WelcomeScreen, LearnScreen and TestScreen each has 2 dependencies. It seems possible that the dependencies due to Tutor's static variables have been interpreted as being in the opposite directions. The 3 dependencies attributed to Tutor by MOT would then be split between the other three classes and the results would agree. However, without knowing which dependencies MetricOne believed to exist, this is only conjecture.

It was argued by Churcher and Shepperd (1995a; 1995b) that counting rules for measures should be more precisely defined. They gave examples of the possible variations in results that could result from different interpretations of the counting rules for the Chidamber and Kemerer (1991; 1994) metrics. In a reply to this criticism Chidamber and Kemerer (1995) implied that these problems were largely synthetic and the 'correct' rules could be implied by the application of simple common-sense. An unexpected outcome of the comparison between the results produced by different measurement tools is the degree to which changes in the interpretation of counting rules can effect measurements. The producers of these tools have no axe to grind in this argument and would seek only to produce accurate measurement and yet they manage to take different interpretations of quite simple measures. This result supports Churcher and Shepperd's position that counting rules should be precisely and explicitly described.

4.8.2 Assessing the functionality of the tool

In the requirements section, some preliminary ideas for an ideal measurement tool were discussed. It has already been mentioned that, within the constraints of the project (and possibly reality), attempting to develop such a tool in full was not feasible. The aim of the tool development was to investigate the feasibility of a subset of these ideal requirements. The development and testing of the tool has shown that user measurement definition, cross-source measurement, and per project metric selection are all feasible. To this extent the tool development can clearly be regarded as a success. However, progress in automating software measurement has not stood

still during the course of this project. How does the functionality of MOT compare to other current measurement tools?

A summary of the results of a comparison of MOT against some of commonly used current measurement tools is given in table 4-3. Of the other four tools used three are commercial tools and the other (JMetric) is a freeware tool produced at Swinburne University. The tools are compared against a superset of their major features. Using this set of features the MOT tool compares very well against the other tools. However, in selecting the set of 'major' features it is difficult to avoid subjective ideas of what the important features of a measurement tool are. Since the author's ideas are themselves the drive behind the design of MOT, the comparison may be unintentionally biased. This is difficult to avoid without an externally imposed comparison framework. As the author is not aware of any existing framework in the literature, the feature set below has been used, but keeping the caveat of possible bias in mind.

The MOT tool has two major omissions from the list of features. It does not allow the users to specify threshold values for measures. This means that it does not give the user any assistance in identifying 'significant' values in the results. The omission is due to time limitations rather any thoughts that this is not a desirable feature. Helping the user interpret the results is an important part of a measurement tool. Indeed, assistance beyond simple threshold detection is desirable and ideas for further assistance will be presented in the 'future developments' section of this chapter.

Another notable omission from MOT would appear to be a hierarchical structure for presenting measurement results. The other tools use this device as a mechanism for adding structure to the results and breaking-up the results into manageable 'chunks'. MOT uses a different device for chunking results. It allows separate sets of measures to be defined for specific tasks. This allows smaller groups of related measures to be collected at one time, rather than collecting the tool's entire catalogue of measures. Having said this, using an entity hierarchy to give metric sets additional structure may also be helpful.

Feature	MOT	Metamata	RSM	JMetric	Metrics -One
Input source-types	Java source Java class Rose	Java source Java class	C C++ Java	Java source	Rose
Can define and save sets of source documents as projects	YES	NO	YES	YES	N/A
Mixed source-type projects?	YES	NO	YES	N/A	N/A
User definable measures	YES	NO	NO	NO	NO
Parameterised measures	YES	YES	YES	YES	NO
Can maintain multiple metric suites	YES	NO	NO	NO	NO
Change over time measurement possible?	YES	NO	YES	NO	NO
Ad hoc investigation possible?	YES	NO	NO	NO	NO
Hierarchical measurement structure (package, class, method, etc.)	NO	YES	YES	YES	YES
Has user definable result thresholds	NO	YES	NO	YES	YES
Result output formats	Table Delim.Text HTML	Table Excel HTML Graphs	Text HTML	CSV Table Graphs	CSV Excel
Number of metrics available	76[†]	13	51	46	37
Number of OO specific metrics	74	10	16	30	35

Table 4-3: Measurement tool feature summary

A conscious omission from the tool was the ability to produce graphical representations of the measurement results. Although graphs of the results are helpful in interpreting results, the ability to produce graphs from data is already widely available and well developed in statistical packages and spreadsheets. As long as the results can be imported into a spreadsheet package there seems little point in duplicating functionality spreadsheets already provide.

There are also some notable features in the other tools. RSM has a very low fraction of OO specific metrics. This stems from its applicability to C as well as OO languages. The usefulness of trying to collect a common set of measures from such widely differing sources might be questioned (especially when considering the earlier discussion on the need for OO specific measures). Although the MetricsOne tool has

[†] This is the number of metrics currently coded in MOT (although there is no theoretical limit).

threshold setting, it only allows the setting of upper bounds. This means that the tool cannot highlight problems shown by metrics with a high-pass or band-pass nature (Abreu, Goulao et al. 1995; Abreu and Melo 1996).

4.8.3 Lessons learned from the tool development

A number of questions were raised and potential problems identified during the development, testing, and use of the tool. The first of these concern the content and structure of the design data that is stored by the tool. It has become apparent that additional information may be useful in the data model. This would allow additional measures to be defined. One interesting possible addition would be additional information on the kind of classes being analysed. This could be considered in multiple dimensions, e.g., class source (project-specific / reused / library), or class function (GUI / problem domain / utility). Classes from these categories are likely to have different metric profiles and the ability to separate them when analysing results could lead to a richer source of design information.

The next set of considerations concern the definition of measures. The datastore currently used by the tool is based on relational technology, which has a number of advantages that have already been outlined. However, there are some associated difficulties. Standard SQL queries cannot deal with recursively defined measures (Oracle's 'connect by' statement can achieve this to some extent, but is not standard SQL). The calculation of the DIT metric would normally require the use of recursion in traversing the inheritance tree. This problem was sidestepped by adding entries to the generalisation table relating each class with each of its ancestors. In effect, this moves the recursion into the mapping and out of the measurement calculation. This is possible with inheritance because the inheritance graph forms a DAG. A measure that was recursively defined on the call graph could not be handled this way because of the possibility of cycles in that graph.

The provision of different source-type mappings introduces a potential problem for the interpretation of the measurement results. This occurs because the different

source-types contain different levels of detail in various aspects of their design information. A consequence of this is that different mappings leave different elements of the data model unpopulated. This means that some measures may not be calculated correctly. Presently, it is not easy for the user to know whether the results are affected by a lack of data, or are a true reflection of the design. It may prove possible to block the use of measures on projects that use mappings that do not populate the entities required by the measure.

One further general comment that has been made by users of the system is the need for different interfaces for different types of users. If the user is interested only in applying existing metrics or metric sets, they may wish other functionality to be hidden from them. This would allow the interface to be optimised solely for their tasks. This request is understandable, as the domain analysis for the system did identify a number of distinct potential user groups, but this change will add greatly to the complexity of the GUI design for the system.

4.9 Future developments

As has been stated earlier, the tool was intended to implement only a subset of the desirable features of a measurement tool. Future developments will be aimed at extending the range of applicability of the tool and its feature set. Increasing the tool's range of applicability means developing addition source mappings. The obvious extension to the system would be a mapping to allow C++ systems to be measured. Additional CASE tool mappings could also be developed.

Currently, the tool only has a simple 'place-holder' measurement method. The development of fully functional components to cover measurement methods, such as GQM, ISO9126 and Whitmire's measurement framework (Whitmire 1997), will be central to the further development of the tool. This should further extend the range of facilities offered by measurement tools, as there are currently no measurement tools that integrate this type of facility.

One of the omissions from the MOT tool that was highlighted in the comparison with the other tools was the absence of threshold setting. Providing guidance for the interpretation of measurement results is certainly important, but simple threshold setting may not be sufficiently flexible. The author proposes extending and generalising this basic guidance for interpretation. Hard, pre-set thresholds are not always applicable. For example, interest may be in relative values, entity ordering or result clustering. A range of potential methods for aiding the interpretation of results should be provided. Another consideration is that with the introduction of measurement methods the interpretation of the results must be done within the context of the measurement method. For example, if we produce a metric set using a GQM method, we would interpret the results by seeing which goals had been achieved.

In order to use measurement successfully the nature and usage of the measures needs to be understood. It might be a beneficial facility to allow the application to connect to an online metrics catalogue to facilitate up-to-date information on available measures. Work has already begun on the development of this catalogue and the tool is already capable of adding new measures downloaded from the web.

4.10 Summary

This section has provided an overview of the MOT software measurement tool produced as part of the author's Ph.D. project. It was shown that it was necessary to build this tool to provide proof of concept for some novel measurement tool features. A preliminary set of requirements for an ideal tool was presented and a subset of these features selected for the initial development on the tool. These features included user definable measures and measure sets, and the ability to measure different sources of design information. A high level description of the architecture of the tool was given along with a description of some of the techniques and tools used in the implementation. An evaluation then showed that the tool produces accurate measurement results and has a favourable set of features in comparison with both research based and commercial measurement tools. This comparison also provided supporting evidence for the view that, when measures are defined or presented, precise and explicit counting rules should be given.

Despite the favourable comparison with other tools there is clearly more work that can be done on the MOT tool and a description was given of some of the major improvements and additions which are planned for the tool in the future. These improvements would include taking input from C++ source code, adding components for measurement methods such as GQM, and providing users assistance in interpreting measurement results.

Blank

In

Original

Chapter 5 The empirical study

This chapter describes work from an empirical study involving the use of measurement to aid the evaluation of a small OO application. The chapter is structured as follows. The first section describes the previous empirical work in the area of design evaluation. Section two gives the rationale for the choice of research technique and outlines the design of the study. Details of the actual conduct of the study are presented in section three. Then results from the study are discussed in section four. Finally, section five summarises these results, attempts to draw some conclusions from this work and points toward possible future work.

5.1 Related empirical work

This section describes other empirical work in OO metrics and examines its relationship to the work presented in this chapter. There are many pieces of empirical work in the area of OO metrics. These studies mainly fall into three types: measurement validation; building prediction systems; and measuring or evaluating particular OO structures or attributes.

As previously discussed, empirical validation in software metrics is not usually the validation of the direct measures themselves. It is the validation of models using these measures. Theoretical validation should show that a measure correctly captures the attribute it is intended represent. Empirical validation shows that this representation can be put to some practical use. This often means using correlation techniques to show that the measure is an indicator of some external measure (e.g. size, fault proneness). Examples of this type of empirical work are often presented along side the definitions of the measures by their original authors, e.g., (Chidamber and Kemerer 1994; Liu, Liang et al. 1994). Occasionally, this type of validation is also performed by third parties, e.g., (Basili, Briand et al. 1995; Harrison and Nithi 1996).

A common type of empirical work in software measurement is the building of prediction systems from measurement values. Typically, size or cost is the attribute being predicted, but quality factors (or their directly measurable proxies) can also be the goal of such prediction system. Quality factors for such prediction systems that have been developed include reusability (Reyes and Carver 1998), maintainability (Cartwright and Shepperd 1997; Li and Henry 1993b) and fault-proneness (Briand, Daly et al. 1998a).

Another example of empirical investigation is the use of experimentation to evaluate particular constructs or issues in object-orientation. Banker, Kauffman et al. (1993) and Melo, Briand et al. (1995) has investigated reuse. The benefits (or otherwise) of using inheritance in OO systems has been assessed in studies by Bieman and Zhao (1995), Daly Daly, Brooks et al. (1995; 1996) and Cartwright (1998). Li and Henry (1995) and Rajaraman and Lyu (1992a) have published empirical studies of maintainability.

In the software measurement chapter the various types of use for metrics were listed. This discussion included the classification scheme for measurement uses proposed by Whitmire (1997). In this scheme there were two types of measurement use that related directly to product evaluation, namely assessment and comparison. Although Whitmire proposes this framework, he provides no empirical corroboration for it. The general literature in the metrics area is also lacking in empirical studies demonstrating the utility of proposed measures in actually aiding the design process.

Lorenz and Kidd (1994) propose thresholds for their measures that could be used to aid design, but they give no empirical support for this utility. Abreu et al. also propose thresholds for the MOOD metrics (Abreu, Goulao et al. 1995), but no empirical support is presented to show whether these actually aid design. Help for designers usually comes in the form of model checkers which search for syntactic errors (Belkhouche and Chavarro 1995) (limited verification), but do not aid the evaluation of design quality.

As far as the author is aware the only existing empirical work in the area of using measurement to aid the design process is the work done at Nottingham University by Gibbon (Gibbon 1997; Gibbon and Higgins 1996). In this study the effects using measures and heuristic thresholds within a CASE tool were studied with reference to the subjects uptake of OOD concepts. This study focused on the education effects of adding measures and heuristics to the design process rather than the effects on the quality of the finished design.

The distinct aims of the proposed study make it a novel piece of empirical work. If no other researchers have attempted a study with these aims it cannot be a replication study. Therefore, a new study design must be designed from scratch.

5.2 Designing the study

5.2.1 Choosing a research technique

There are many forms of empirical research covering both quantitative and qualitative investigation. The most commonly used forms in software engineering are survey, experiment and case study (Wohlin, Runeson et al. 2000; Pfleeger 1994) (action research (Greenwood and Morten 1998) and ethnography (Creswell 1998) are also occasionally used). The question to answer is which type of empirical work will best support the study's objectives?

A survey is research 'in the large', that is it tries to assess a topic across a broad range of companies or individuals. There is no attempt to control variables in surveys, largely because surveys are typically retrospective. A survey requires that there is a population of subjects to poll who are already familiar with the area of investigation. For a survey to be a suitable research technique there would need to be an existing base of subjects who have experience of using measurement for OO design assessment. No such base of measurement users is available. Consequently, a survey would not be a viable research technique for this study.

Experiments are formal, rigorously controlled investigations. The level of control required means that an experiment typically involves a 'toy' system and a small number of people. Experiments can, therefore, been regarded as 'research in the small'. In experiments, one or more variables are manipulated and the outcome on some other variable, or variables, of interest is observed. The intention is to separate the effects of the individual inputs (Fenton and Pfleeger 1996). Experimentation would be a viable technique for this study.

Case studies have been described as research in the typical. They look at a small number of cases (often only one). Case studies are planned investigations into a particular entity within its environment. They involve the carefully planned collection and analysis of data but do not use the systematic variable manipulation found in experiments (Yin 1994). A case study would also be a viable technique for this study.

Ethnography involves a longitudinal study based primarily on direct observations and interviews. The focus of such studies is interpreting the behaviour within a cultural or social group. This type of research may be used within software engineering for analysing group dynamics within collaborative work. Since the focus of the proposed study does not involve group work, this method is not a suitable technique.

Action research is an approach where the researcher participates in the system under study. It also actively encourages members of the organisation being studied to participate in defining the problems to be examined and in collecting and generating data pertinent to these problems. An action research approach aimed at the goals of the proposed study would require an organisation in which there was a perceived problem in design assessment. The action researcher would then enter the organisation and, aided by interested individuals in the organisation, would seek to find a solution to this problem. One possible difficulty with this approach is that the direction the research takes is not entirely within the control of the researcher - the organisation may decide to seek a solution in different direction to that the researcher originally envisaged. It is also the case that no suitable organisation was available with

which to perform such a study.

Of the original five research techniques, all but two have been considered unsuitable for this study. The remaining choice is whether to use an experimental or case study approach. A point that has a major bearing on this choice is the amount of preparation the subjects will require before being able to make an informed use of measurement in either of these tasks. It would seem unrealistic to expect volunteers to undergo this amount of training (except under levels of bribery outside the resources of this project!). A solution to this problem would be to use students as subjects and to make the preparation part of the course.

There is also a similar problem with the amount of time necessary to perform an assessment or comparison with even 'toy' systems. If bribery were again abandoned as a means of persuasion, the obvious solution would be to set an assessment and/or a comparison as course work.

If this rationale is followed, further considerations come into play. An experiment would require multiple treatments. There are two obvious ways of approaching this problem. The subjects could be divided into two groups, each of which would effectively be doing a different assignment. Alternatively, a within subject design could be used, in which two groups of subjects each tackle two tasks. Each of these options has serious drawbacks, either from a pedagogical or ethical perspective. Having different subjects tackle different problems makes comparative marking between the groups very difficult. A within subject design introduces ordering effects into the performance of the tasks. This makes both assignment marking and data analysis more difficult. It was also felt that having the subjects tackle one larger task would have more pedagogical value than tackling two smaller tasks. This meant the decision was to set a single task for all subjects. Perforce, this means a single treatment and therefore rules out a formal experiment.

Another factor against the use of a formal experiment is the immaturity of the topic

under investigation. There has been very little work concerning how measurement is used to support design assessment. When there is little knowledge available on the topic it is difficult to design a formal experiment. Without a good knowledge of the influencing factors or their relationships, formulating a hypothesis is difficult. In addition, designing the experiment to control the effects of confounding factors becomes almost impossible.

Case studies usually involve investigating a phenomenon in its 'natural setting'. Normally, a case study would involve investigating a design review in a real industrial setting on a real project. The task being studied here is performed in what is approaching laboratory conditions. The study is being carried out in an academic rather than an industrial setting and the project being studied has been selected specifically for the study. Yin (1993) describes a methodological approach midway between an experiment and a case study that he refers to as a quasi-experiment. Quasi-experiments are possible where there is insufficient control over variables of interest for a formal experiment. Unlike case studies, quasi-experiments require a sufficient number of subjects to produce multiple data points and a limited number of variables of interest.

As mentioned previously, the topic under consideration is fairly immature. This tends to lead to exploratory studies with potentially large number of variables under consideration. However, this study does have a sufficient number of subjects for a quasi-experiment. There are some factors in the topic and organisation of this study that would indicate that it is case study and others that it is a quasi-experiment. The name used to label the type of study seems of little importance as long as the nature and limitations of the study are clearly understood. For the remainder of the chapter, this empirical work will be referred to as a case study.

To recap, the decision was to perform a case study with a group of students who would receive tutoring in design measurement and its application during their course and would perform the case study as an assignment.

5.2.2 The study's questions

Practical constraints have led to the decision that the research technique used was to be a case study. The next point to consider is what questions this study will attempt to answer. The immature nature of this field suggests the use of an exploratory style case study. According to Yin (1994), one of the strengths of case studies is in answering exploratory style 'what' questions or descriptive 'how' and 'why' style questions. It seems sensible to phrase the study's questions to take advantage of these strengths.

Possible questions:

1. What support is necessary to enable the practical application of design measurement?
   - What additional support is desirable to improve the application of design measurement?
2. In practice, how would engineers use measurement results during design evaluation?
   - Which elements of the analysis can be aided by the use of measurement?
   - Which elements can't?
3. Given the necessary training and experience would engineers choose to apply measurement to help design systems?

5.2.3 Unit of analysis

The next consideration is the unit of analysis. Here we have to decide, what is the case we're studying? The overall goal of the study was to see how designers would use measurement results in practice and what type of support they would require. Different design activities might well require different kinds/levels of support and may find the use of measurement be more or less beneficial. This would suggest treating design assessment and design comparison separately, enabling separate and distinct findings for each type of activity. However, as previously mentioned it was necessary for all subjects to perform the same assignment. If the study was to cover both assessment and comparison, each subject must undertake each of these tasks. There are therefore, two sets of cases, a set of assessment cases and a set of

comparison cases. One assessment is also paired with one comparison because they are done by the same subject. This pair of cases is best analysed together when considering propositions that deal with the approach and attitude of the designer.

The design for the case study will therefore have a unit of analysis that is the individual assignment. Each assignment will be considered as consisting of two embedded units, an assessment unit and a comparison unit.

5.2.4 Data collection techniques

Yin lists six possible sources of evidence for case studies. These are documentation, archival records, interviews/surveys, direct observations, participant observations and physical artifacts.

1. Documentation

   One of the data collection techniques to be used is to require the subjects to produce a report documenting their work on the task.

2. Interviews / surveys

   A questionnaire will also be used to elicit background and attitudinal information from the subjects. It will also be used to provide corroborative data to backup the report data.

3. Participant observation

   After the performing the task the subjects will be invited to give their opinions of measurement-based assessment. This will be done as part of the questionnaire.

4. Direct observation

   Direct observation of the actual assessment task is not really practical. The length of time spent on the task (nominally 25 hours) would make this unrealistic. However, it will be possible to monitor the subjects' data collection by adding automatic action logging to the design measurement tool they will be using.

5. Archival reports

   There are no archival reports available to use.

6. Physical artifacts

   There are no physical artifacts involved in the field of study.

5.2.5 Design of debriefing questionnaires

The purpose of the questionnaire was to gather attitudinal data not available from the other data collection techniques and also to gather information to triangulate with data gathered from the other sources. Additionally, questions were included to check whether any elements of the design of the task had adversely affected the results from the case study.

5.2.5.1 Information to be elicited

The information required from the questionnaire can be split into three sections, before, during and after performing the task. The 'pre-assignment' questions are to collect information on the subject's background and experience. More specifically, they are to determine the amount of experience the subjects have in programming and in software design. The 'doing-assignment' questions are to gather information relating to the conduct of the task itself. Information required from this section included:

- time and effort spend performing the task;
- how the subjects went about selecting metrics to apply;
- how the subjects used the data sources describing the system under study;
- the subjects' views on the individual metrics they used;
- the subjects' use of the measurement tool.

The 'post-assignment' section was used to check for any changes in attitude towards using measurement following the subjects experience whilst performing the task.

5.2.5.2 Question design

There are many texts giving guidance on the design of the questions themselves, e.g., (Fowler 1995; Payne 1951). Other questionnaire and survey texts also give guidance on the layout and presentation of questionnaires, e.g., (Berdie, Anderson et al. 1996; Fink and Kosecoff 1985). In designing the exact wording and layout of questions, these texts describe a large number of considerations and relevant advice. The questionnaire was designed with these recommendations in mind and was improved through review and redesign. The final version of the questionnaire can be see in

appendix G.

5.2.6 Validity

In judging the quality of a research design Yin (1994) lists four tests. These tests are construct validity, internal validity, external validity and reliability.

Construct validity involves establishing that correct operational measures have been used for the concepts being studied. One suggested way of establishing construct validity is the use of multiple sources of evidence (as was done with this case study).

Internal validity involves establishing a causal relationship, whereby certain conditions are shown to lead to other conditions, as distinguished from spurious relationships. This case study is largely exploratory and descriptive in nature. Yin (1994) describes internal validity as being appropriate only for explanatory or causal studies.

External validity involves establishing the domain to which the study's findings can be generalised. The use of multiple case designs is one way to show external validity. A consistency of results between different cases suggests that the findings can be generalised beyond a particular case. However, multiple case designs are intended for replication, rather than sampling. This means that no inferences should be drawn about the population to which the results will generalise. When dealing with exploratory case studies the only generalisation that can be done is to theory. It then requires an experiment to confirm the theories and assess their range of generality.

Reliability involves demonstrating that the operations of a study - such as the data collection procedures - can be repeated, with the same results. In this study defined protocols were used for extracting the data from the subject reports. Where subjective judgements about design assessment were required multiple 'experts' were used. The use of a case study database should also help ensure consistent data collection and storage.

5.3 Conducting the study

This section describes the actual running of the case study. One section notable by its absence is the preliminary study. The original intention was to run a preliminary study. However, as the case study was being run as an assignment, it was fixed in time by the teaching calendar. The preparation of the study's materials took longer than expected and there was insufficient time available to organise a preliminary study.

This section commences with a description of the subjects chosen for the study and analyses their previous levels of experience. The preparation the subjects received is then discussed, followed by a description of the case study task. Finally, the procedures for data collection and extraction are covered.

5.3.1 The subjects

The subjects selected for this case study were those taking a unit at Bournemouth University entitled 'Object-oriented software engineering' (OOSE). The unit was selected because it is an advanced unit (presuming some prior knowledge of OO), also, the necessary preparation and task itself fitted well into the remit of this course. Originally there were 16 subjects, 9 were full-time final year B.Sc. students and 7 were part-time students working full-time in IT. One of the students has been excluded from the analysis, since the subject failed to complete the task in the manner requested, leaving 8 students and 7 industrialists.

Figure 5-1 gives some idea of the subjects' OO experience prior to the OOSE course. It shows the maximum levels of experience of the subjects in any OO language and in any OO design method. Almost half of the subjects had used an OO language professionally and all but two had prior experience of at least the level of a formal taught course. There is generally less experience of OO design methods. Only two of the subjects had used any of these professionally and three had not received any prior formal training in OO design.

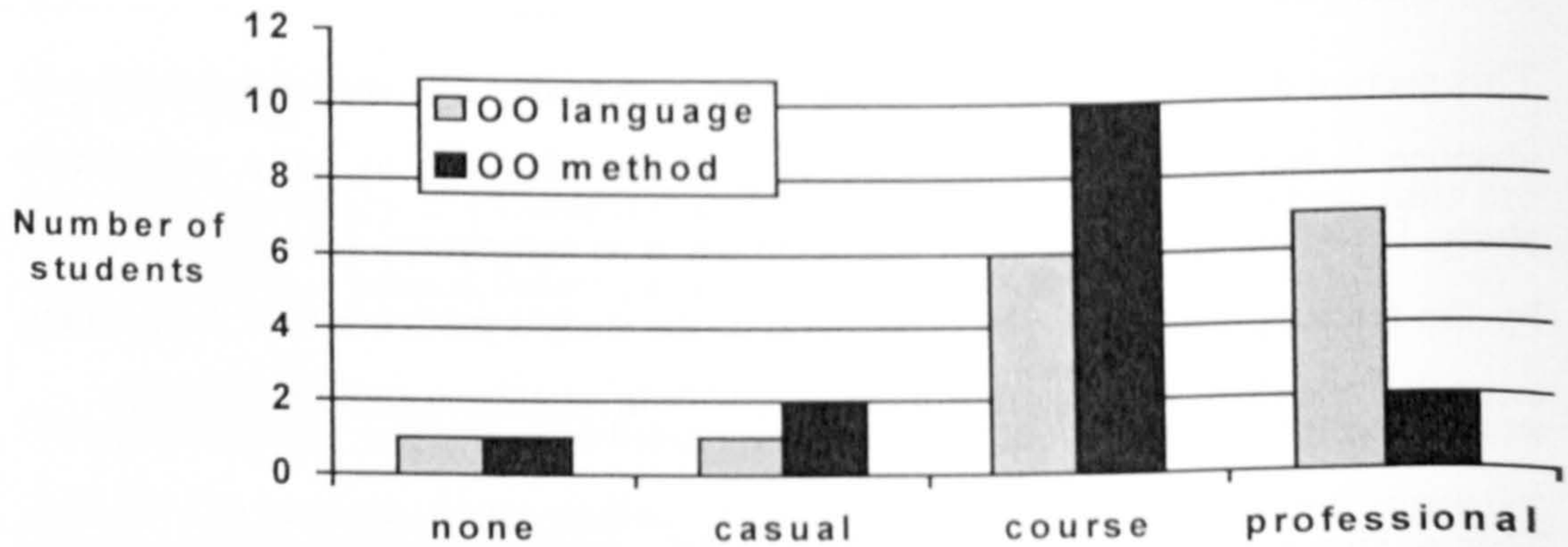


Figure 5-1 Subject's prior experience in object-orientation

Figure 5-2 helps to give a more balanced view by showing the subject's experience of non-OO development. By comparing Figures 5-1 and 5-2, it can be seen that the subjects generally have more experience of OO design than structured design methods. However, more of the subjects have used procedural languages than OO-languages in industry.

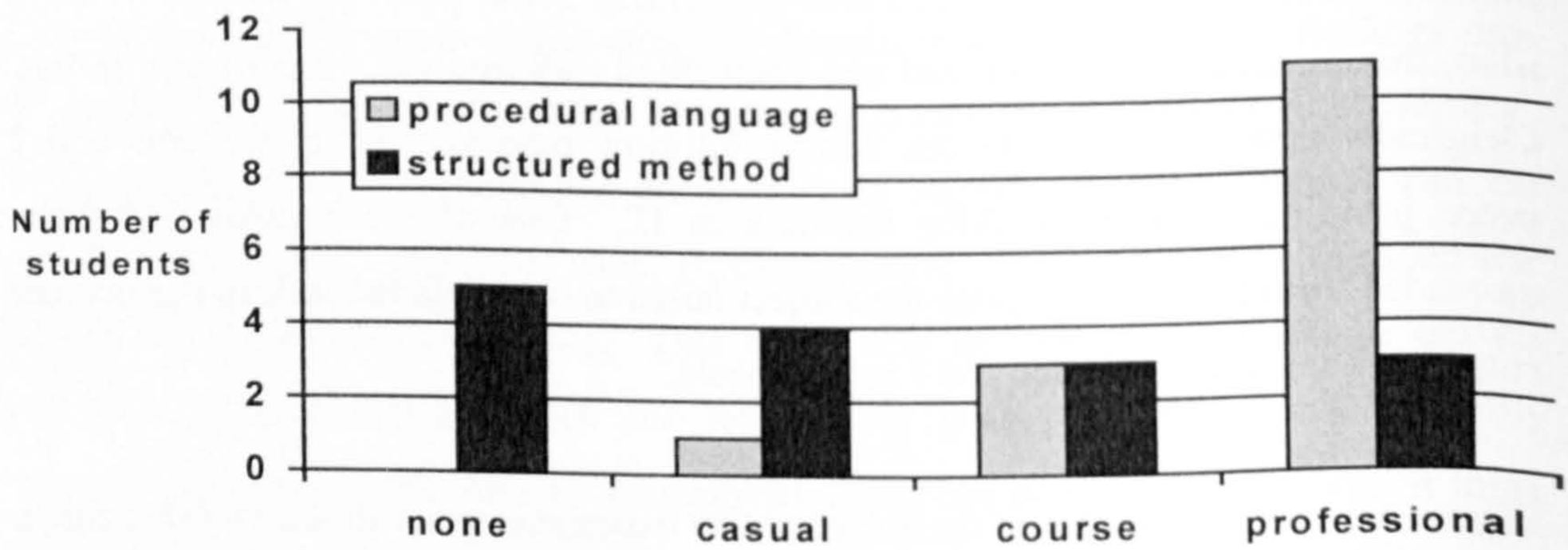


Figure 5-2 Subject's prior experience of non-OO development

It may also be worth noting, when comparing figures 5-1 and 5-2, that object-orientation is usually considered to have a steeper learning curve than traditional techniques (Fichman and Kemerer 1997; Vessey and Conger 1994). This may mean that, after equivalent levels of exposure to OO and procedural methods, subjects may have a lesser grasp of OO.

Figure 5-3 shows the subject's prior experience in Java and UML. These are

particularly relevant as the design for the case study was presented in UML and the system coded in Java. In can be seen that there was little knowledge of UML in the group prior to taking the OOSE unit, although all but 2 subjects had some prior experience with Java.

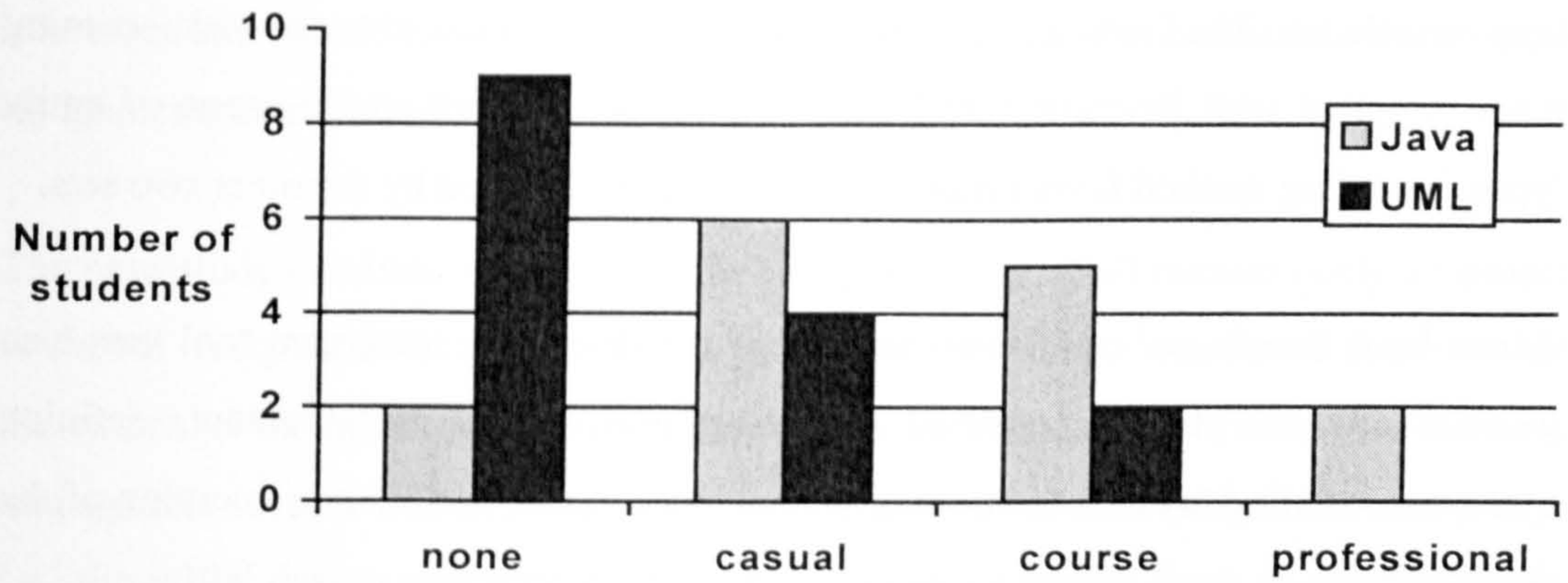


Figure 5-3 Student's prior experience of Java and UML

5.3.2 Preparation

As part of the OOSE course the subjects studied OO programming in Java, design presentation using the UML, design evaluation, design patterns, and the process of OO development. The work on design evaluation covered both heuristic-based evaluation and OO design metrics. The subjects received 3 one-hour lectures on heuristics and metrics and they spent an equivalent amount of time on related paper reviewing and focused discussion.

The heuristics material was based on our own review of OO design heuristics (Kirsopp and Webster 1997) and the book by Riel (1996). The subjects were introduced to the idea of heuristics and the issues surrounding the use of heuristics. They were also talked-through a representative subset of the heuristics from the catalogue.

The metrics teaching material focused on the work of Chidamber & Kemerer (CK) (Chidamber, Darcy et al. 1998; Chidamber and Kemerer 1994; Chidamber and Kemerer 1991) and Brito e Abreu (Abreu and Melo 1996; Abreu, Goulao et al. 1995;

Abreu and Carapuca 1994), as well as covering the Lorenz & Kidd metrics (Lorenz and Kidd 1994). The CK metrics were chosen because they are the most widely cited and validated set of measures. The MOOD metrics of Abreu et al. are also widely cited and have received third party validation. The measures in the MOOD set are also focused on a different entity type from the CK metrics, being system-level rather than class-level. The Lorenz and Kidd measures were well suited to the task - coming ready supplied with heuristic thresholds. They also covered a wide range of entity types, including method level measures, which are not covered by the other two sets.

A one-hour 'hands-on' tutorial on the use of the design measurement tool was also given. The tool tutorial covered setting-up projects for measurement, defining measures, creating measurement sets, collecting measures, and ad-hoc querying of the design model.

5.3.3 The task

The application used for the study was the 'Second Language Vocabulary Tutor' (Vocab Tutor). This is a small application to allow learners of a second language to increase their vocabulary through a process of actively learning new words and testing themselves on their progress. The system was developed by the OOSE unit leader as a teaching aid for OO concepts and programming. It is coded in Java 1.1 and has a GUI interface built using the standard AWT (abstract windows toolkit) components. The Vocab Tutor consists of 15 classes and around 1500 lines of code. Five of the classes make up the GUI and the remaining ten implement the problem domain. The application contains a simple inheritance hierarchy with a depth of one - a single parent class with a number of child classes.

The subjects were provided with design documents (appendix H) and source code (appendix I) for the Vocab Tutor. They were also given access to the MOT design measurement tool, which allowed the collection of metrics from the suites suggested by Chidamber and Kemerer (1994), Lorenz and Kidd (1994) and Abreu and Melo (1996).

The task comprised two major sections. In the first section, the subjects were asked to assess the design of the Vocab Tutor. They were asked to indicate the weak points in the design and to back up those arguments with heuristics and metric results. In the second section they were to outline the design changes they would make and explain the motivation for these changes, again they should back up these arguments with heuristics and metric results. The exact wording of the task can be seen on the original assignment sheet included as appendix J.

The case study database is very complex being made up of 20 entities (only an aspect is shown in figure 5-4). The case study also generated large amount of data, so that the extent of the tables is also very large. To keep the amount of data analysis to a manageable level it was decided to restrict the analysis to first half of the case study, i.e., the initial design assessment. No discussion of, or results from, the redesign and reassessment section of the task will be presented in this thesis.

To give an idea of the amount of work involved in performing the task the subjects were asked to estimate (after the event) the number of hours they spent on different aspects of the task. Table 5-1 shows a summary of this information. The table shows the total time and a breakdown of the time spend on the three main sub-tasks. It should be noted that the times given cover both the assessment and re-design sections of the task.

Variable	Mean	Median	Min	Max
Time on research	6.7	7	2	12
Time on analysis	6.3	5	2	16
Time on write up	9.3	8	4	16
Total time	22.2	21	9	42

Table 5-1 Summary of time spend on the task (in hours)

5.3.4 Data collection

The data collection for this case study comprised of three different elements. Firstly, all actions performed using the design analysis tool were logged. Secondly, each

subject was required to produce two written reports, one for each section of the task. Finally, after the reports were handed in, the subjects were asked to complete a debriefing questionnaire. The data collected were organised into a case study database.

The data logging activity didn't prove to be as useful a source of information as originally hoped. This was due to the subjects using the tool in a very limited manner. It was hoped that the subjects would create their own measurement sets to suit their own particular goals. It was thought that they would use the tool's ad hoc query facility to follow-up on anomalous results. There was even the possibility that they might create new measures to automatically assess particular heuristics they wanted to apply. What actually happened was that the subjects simply collected results from one or two of the predefined metric sets and performed all of their subsequent work off-line. This meant that no meaningful data was collected from the logging activity. The analysis will therefore be done with data from the reports and questionnaires only.

The reports produced by the subjects had to be codified into a form suitable for further analysis. In considering the assessment of a single design, the main output of the assessment is the set of problems identified. Additional information might include any justification for believing particular elements of the design to be problematic. From the phrasing of the task, these problems are likely to appear as heuristic violations. These violations might be justified using measurement results. As mentioned earlier in the description of design heuristics, a heuristic violation in itself may not represent a problem. If the violation occurs outside of the intended context of the heuristic, the apparent violation may have resulted from a misapplication of the heuristic.

Even within the correct context, heuristics are only guidelines and not 'hard-and-fast' rules that always apply. For these reasons, the author wishes to separate the notions of problem and heuristic violation. The term 'issue' will be used to describe any reported heuristic violation, and 'real issue' to describe a heuristic violation that is believed (subjectively) to indicate a real design problem. Any mention of a heuristic violation in the subjects' reports was entered into the case study database. References to

measurement results and the use of measurement results to justify these issues were similarly recorded. Additional information was also recorded, such as the nature of the heuristic that was violated, the location of the violation, etc.

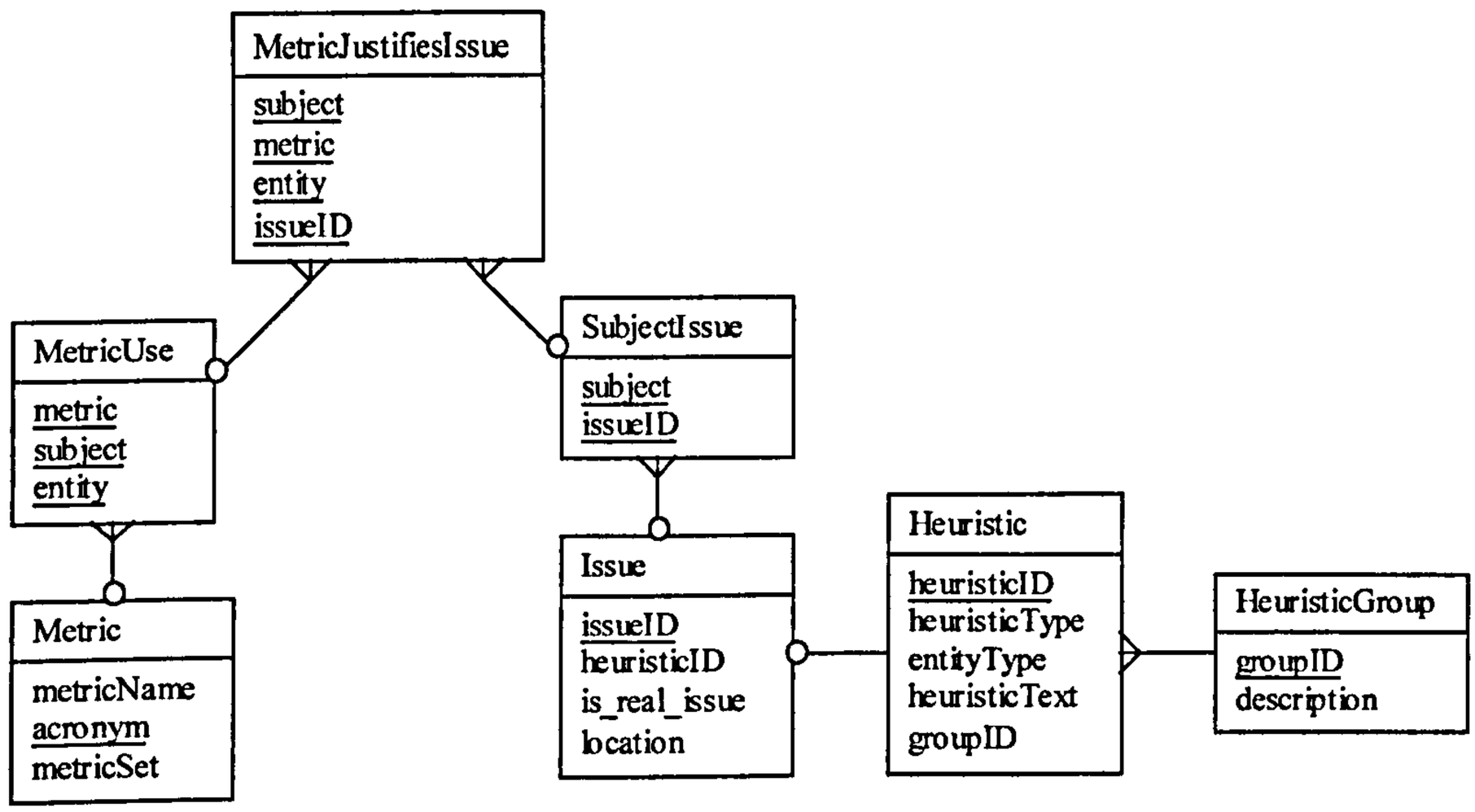


Figure 5-4 An aspect of the case study database model

The data model in Figure 5-4 should help to clarify the relationships between these variables. An 'Issue' is the violation of a particular 'Heuristic' at a particular location and may, or may not, be considered a real issue. A 'SubjectIssue' is the use, by a particular subject, of a particular issue in their evaluation of the system. A 'MetricUse' is a reference by a subject to the measurement of a particular entity in the system using a particular metric. An entry in 'MetricJustifiesIssue' shows that a particular subject justified the inclusion of a particular issue by a particular measurement result ('MetricUse'). Heuristics are also assigned to a particular HeuristicGroup. A heuristic group collects together heuristics from different sources but with similar underlying rationale and intent (this will be discussed further at the point at which it is used in the results section).

The report produced by the subjects described their actions during the design assessment in unstructured prose. Clearly, there will be some circumstances where the mapping of such unstructured prose onto the structure of a relational data model is less than clear-cut. For example, a subject may have stated that a heuristic concerning

class size had been violated but not explicitly state the measure used in their assessment. In such cases a 'best guess' as to the subject's actions and intentions had to be made (in the example above, the most likely measure would been considered to have been implicitly used).

The problems with mapping prose onto the data model were not limited to simply deciding what the subject intended. There are subtleties in the manner in which the subjects used the measures and heuristics that are lost in the mapping. Much time was given to deciding on the structure of the data model. A more complex data model could capture more information, but would be more difficult to analyse. In the end a trade-off had to be made between loss of information and ease of analysis.

An example of some of the problems identified by the subjects and the type of justifications used is given in appendix K. This should help to clarify the nature of the mapping between the reports and the data model.

As described earlier the debriefing questionnaire was designed to collect both information not covered by the other data sources and to add corroboration to the other data sources. It included sections on attitudinal information, background experience, and time expended. This information was also input into the case study database. The full data model for the case study database can be seen in appendix L.

5.4 Results

The results are grouped into seven subsections. These subsections cover the following areas of analysis:

1. Metric selection: how the subjects selected the metrics they used to perform the assessment
2. Metric usage (by metric): the patterns of use for particular metrics
3. Metric usage (by subject): the level of metric usage by the individual subjects
4. Issues: the issues found by the subjects
5. Heuristics: the manner in which the subjects applied heuristics in the evaluation
6. Differences between students and industrialists: evidence for differences in the

approach between the industrialists and the students

7. Subjects' views on measurement: the subjects' post-study attitudes toward software measurement

5.4.1 Metric selection

The first question, which faced the subjects when performing the task, was the selection of the metrics they were to use. How would they go about this selection? The two pie charts contained in Figure 5-5 show the results from two of the questions from the debriefing questionnaire.

1. 'Did you consider taking a pick'n'mix approach to selecting metrics? (taking some metrics from one author's set and some from another)'

2. 'Did you consider using a different set of metrics for analysing the design refinement than you did for the initial assessment?

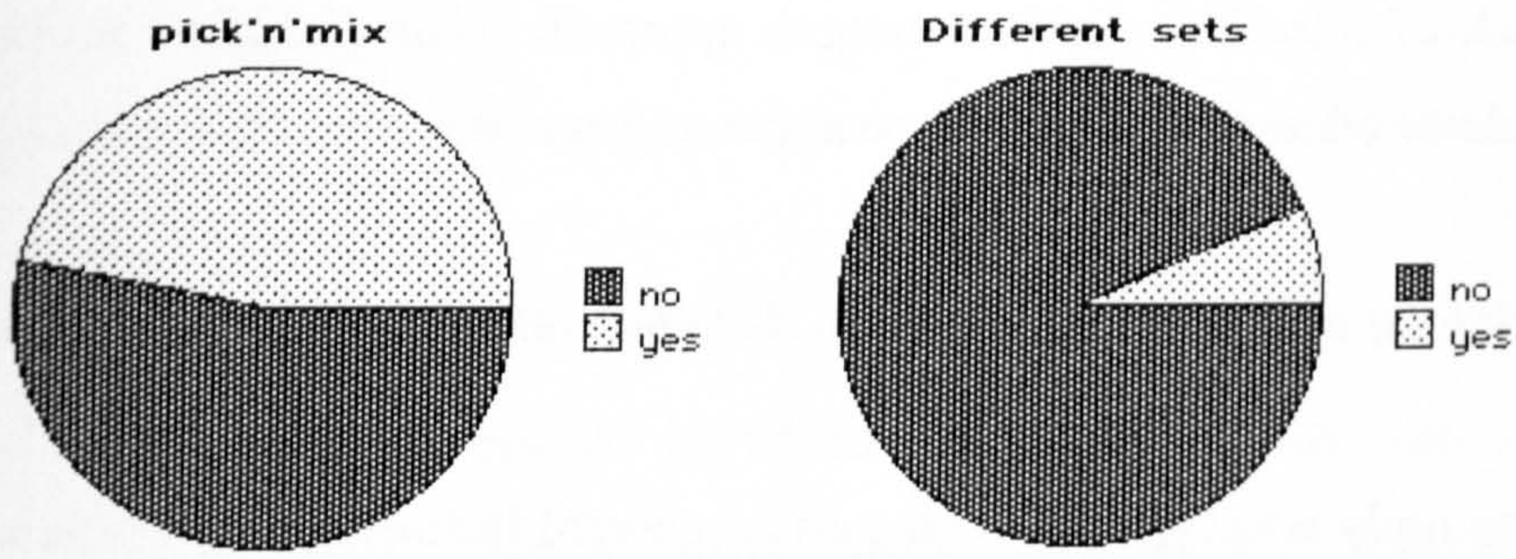


Figure 5-5 Results from the metric selection section of the questionnaire

The results from the first question show that over half of the subjects didn't even consider selecting across suites of metrics proposed by different authors. They simply chose a set, as given, and tried to apply it.

Although this chapter concentrates on the evaluation section of the task, the subjects also had to redesign the system based on their initial assessment, and compare their new version of the system to the original. From a measurement perspective, the nature

of this second task is a qualitatively different task from the first (Whitmire 1997). Firstly, the measurement results are now being used to compare relative 'quality', rather than being used for finding specific problems. Secondly, the measurement results are being compared with similar results from another system rather than a threshold set by a heuristic. Question 2 in this section was intended to discover whether the subjects recognised this distinction and considered using different metrics to achieve these differing ends. The answer, quite overwhelmingly, is that they did not.

If most subjects did not consider selecting metrics from different authors, or using different sets of metrics for different tasks, how did they rationalise their selection? The following are typical answers given by the subjects when asked for a rationale for their metric selection.

'Ran them all and used the ones that best fit the process I was going to use.'

This is more a lack of selection. Such a shotgun approach seems likely to be very inefficient, particularly when scaled up to large applications.

'By ...deciding which set was the most suitable to the application.'

This kind of vague reply was typical. It suggests a general lack of thought given to metric selection.

'The only suite I knew well and understood was CK.'

'The ones I could find information about.'

This type of reply shows the importance of comprehensible and accessible information about metrics. If you can't find information about a metric, or you can't understand that information, you won't use the metric. Accessibility of information seems to have been the main reason for the lack of use of the Lorenz and Kidd (1994) metrics. Papers describing the CK and MOOD metrics were given to the subjects.

The Lorenz and Kidd (L&K) metrics are described in a book and as such were not so easily (cheaply) available.

Figure 5-6 shows the level of use of the various metric sets and combinations of sets. Almost half (7) of the subjects used only the Chidamber and Kemerer metric set. A further 4 used both the CK and the MOOD metrics and one used all three sets (CK, MOOD, L&K). This means that 12 of the subjects used the CK metrics while 7 used the MOOD measures and only one used L&K. One subject didn't used measurement support for their evaluation.

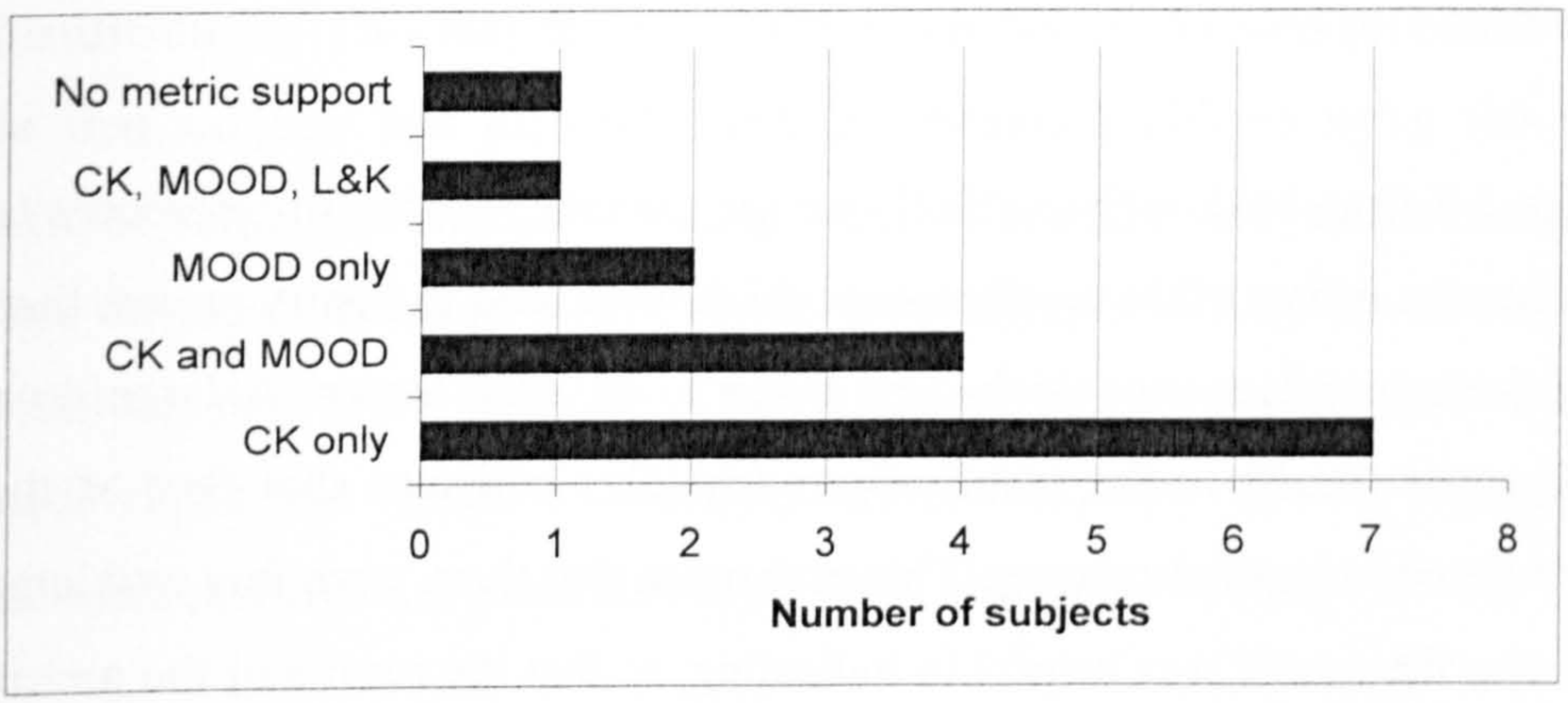


Figure 5-6 Metric-sets used by subjects to support their analysis

5.4.2 Metric usage (by metric)

In this section, the manner in which the actual metrics were employed will be examined. The uses of a metric can be considered at a number of levels. A measurement result from the metric could be referenced in a subject's report. This measurement result may, or may not, have been used to justify a particular issue. The issue may, further more, be considered (by the 'experts'[9]) either to be a 'real' problem with the design or a non-issue. Figure 5-7 displays the uses of metrics at these three levels.

Reading from left to right, the first four metrics are from the Lorenz and Kidd (1994) set (L&K). Each of these metrics was only referenced once and not used to justify an issue. The next six metrics are the MOOD metrics (Abreu and Carapuca 1994; Abreu,

[9] The 'expert judgement' was agreement between the author and the marking tutor for the assignment.

Goulao et al. 1995; Abreu and Melo 1996). The final six metrics are those proposed by Chidamber and Kemerer (1991; 1994) (CK). Appendix M gives a brief description of the metrics shown on Figure 5-7. A point to note is the obvious difference in usage between the MOOD metrics and the CK metrics. This is due to the different nature of the metrics and the way the uses are counted. A use is defined as the application of a particular metric to a particular entity by a particular subject. The entity measured by the MOOD metrics is an entire system. There is therefore only scope for one use of each of these metrics by each subject. The CK metrics, on the other hand, measure classes. This means that, potentially, each subject may use each CK metric with each class in the system.

Reading Figure 5-7 from back to front: firstly we see the total number of references to each metric; then the subset of these references which were used to justify issues; then the subset of these justifications which were taken to be 'real' issues. A large drop from the references (back) to the justifications (middle) indicates that few of the references were used in justifying issues. This may mean that there were few problems in the system that the metric was helpful in indicating, or that the results of the metric were difficult to interpret into concrete problems.

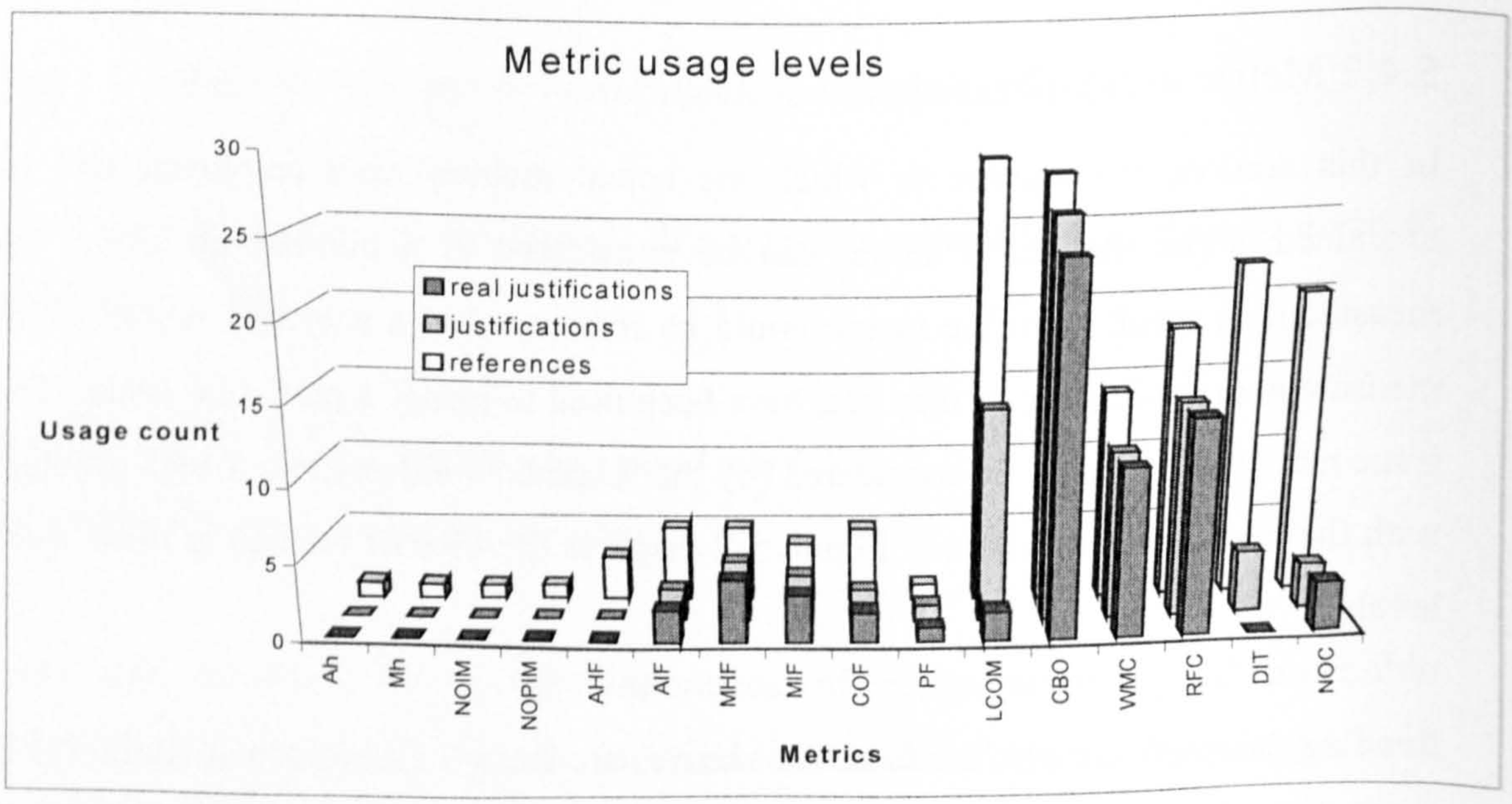


Figure 5-7 Metric usage, by metric and usage type

A large drop from the justifications (middle) to the real justifications (front) indicates that few of the issues justified were considered to be real issues, i.e., the metric results

led to erroneous conclusions (or had been misinterpreted).

Considering some specific examples, CBO and LCOM are both heavily referenced. This indicated that the subjects thought that there were coupling and cohesion issues to be considered within the system. A clear difference between these metrics can be seen in the rate at which their use declines toward the front of the chart. CBO decreases very little from front to back, showing that it clearly indicates particular problem areas to the subject and the problems are identified are consider to be 'real' problems. LCOM on the other hand halves in value between references and justifications. This may mean that there were fewer cohesion problems in the system, or that subjects had difficulty finding cohesion problems using this metric. The LCOM value falls almost to nothing when we consider real justifications. This means that on most occasions where the subjects interpreted LCOM's result as showing a problem, they were mistaken. These apparent problems with LCOM are consistent with views and results of other authors (for example (Basili, Briand et al. 1995; Henderson-Sellers, Constantine et al. 1996; Hitz and Montazeri 1996)).

The DIT and NOC metrics have very similar patterns of use, reasonably high references and then very few justifications. This appears to be largely due to the size and 'shape' of the system. There will be little scope for finding problems using inheritance-based counts in a system containing only 1 level of inheritance (excluding system classes).

The questionnaire included the opportunity for the subjects to express their views about the usefulness and the understandability of the metrics that they had selected. Responses were given on a numeric scale between 1 and 4 (4 being the most useful or understandable). Table 5-2 below shows the median values (metrics on which fewer than 5 opinions were expressed have been excluded).

Metric	Useful	Understandable
CBO	4	4
WMC	4	3.5
RFC	3.5	2
COF	3	4
POF	3	3
NOC	2.5	4
DIT	2	4
AHF	2	4
MHF	2	4
AIF	2	4
MIF	2	4
LCOM	2	3

Table 5-2 Subjects' opinions on the metrics they selected

Note the parallels between this table of subjective views and the analysis of actual usage earlier. LCOM's possible problems again appear with the equal lowest usefulness and the equal second lowest understandability. The subjects did not find DIT or NOC particularly useful for evaluating this particular system although they were easily understood. Low values for usefulness of the MOOD metrics reflect the difficulty of trying to infer specific problems from system wide metrics.

RFC is also an interesting case - being the most difficult to understand whilst also being considered very useful. The fact that the counting rules for RFC are quite involved is likely to have resulted in the low understandability. However, from the manner in which the subjects used the RFC metric, they clearly understood that it was type of coupling measure and thought coupling measures useful. This suggests that complex measures are not necessarily problematic. They may still be successfully applied so long as those utilising them understand the underlying intent.

5.4.3 Metric usage (by subject)

The chart below (Figure 5-8) gives an indication of the range of measurement use by the individual subjects. It shows both the number of distinct measures used by each subject and the number of occasions on which the subject used these measures. The range of distinct measures used by the subject varies from 0 up to 10 and the number

of measurements cited in the evaluation varies from 0 to 33. A low value for metrics used suggests that the subject's assessment utilised mainly semantic heuristics. A high value suggests a more syntactic approach. The ratio of measures used to measurement uses also varies widely. This is mainly due to the entity type to which the measures apply and will be discussed later.

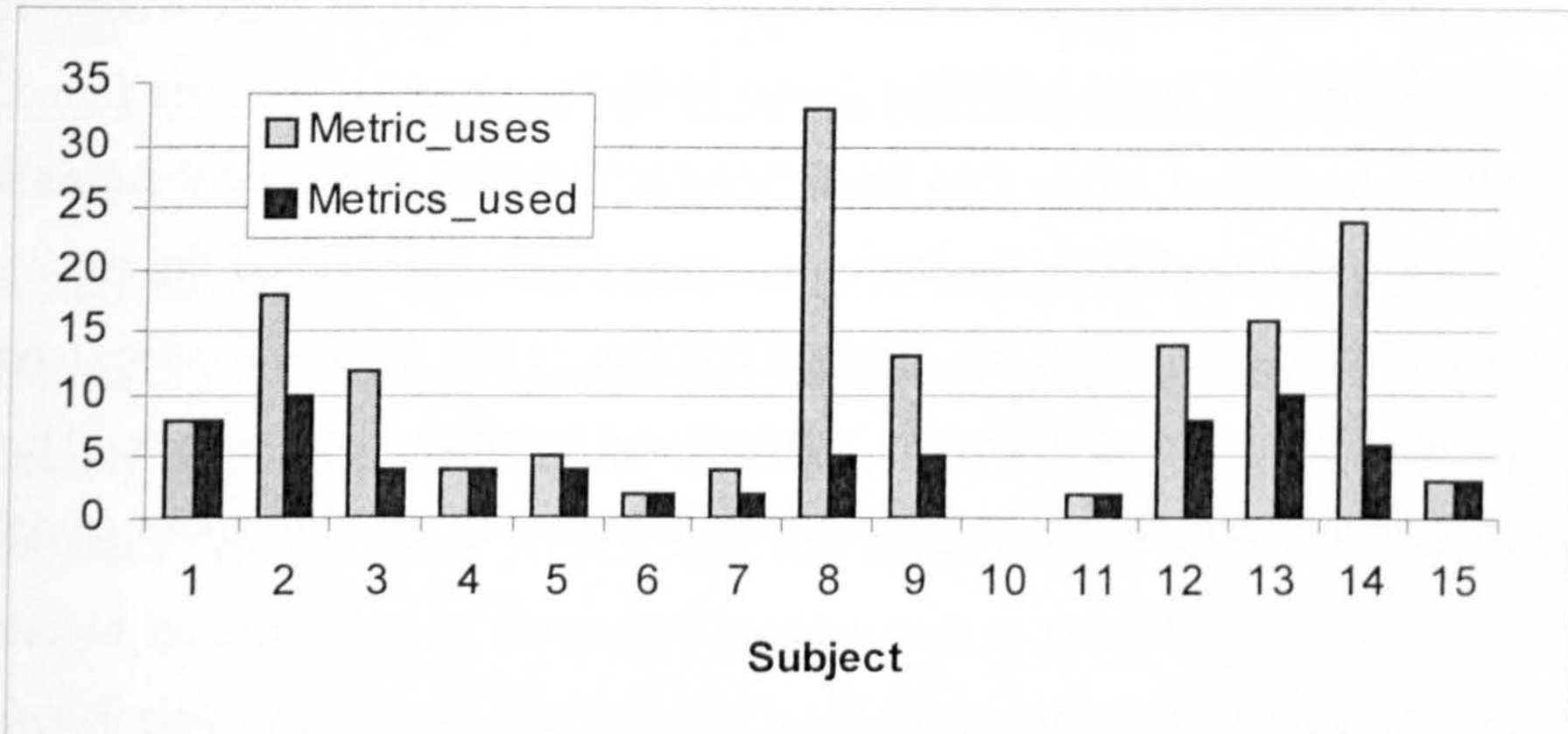


Figure 5-8 Number of metric uses and metrics used by subject

5.4.4 Issues

As previously mentioned, issues are heuristic violations. They can be divided into four quadrants as shown in Table 5-3. The 'found' column contains the number of distinct issues identified by the subjects. Those issues in the 'not found' column are those issues identified by the 'experts', but not found by any of the subjects. The rows divide the issues based on whether the 'experts' considered those particular heuristic violations highlighted real design problems. Only three of the quadrants are used as the number of false issues not found is not assessable (possibly infinite!).

	Found	**Not found**
Real issue	86 40.4%	98 46.0%
False issue	29 13.6%	

Table 5-3 Distinct issues by issue quadrant

Table 5-3 shows that the subjects to have detected less than half of the real issues. This might be taken to suggest that the subjects were poor at finding problems with the system. There are reasons for not taking this reading of these results.

Table 5-4 shows the distribution of issues found and not found with the importance of the issues. This table clearly shows that 67% of issues not found were trivial in nature, whereas 90% (75 found and 9 not found) of the important issues were found. These results indicate that rather than being poor at detecting issues, the subjects simply focused their attention on the more important issues and ignored the trivial. Another factor, which compounds this apparent problem, is that the trivial issues tend to be associated with the more numerous 'method' and 'attribute' entity-types. This tendency exaggerates the failure to identify trivial issues by creating more trivial issues.

	trivial		**significant**		**important**	
	Count	**%**	**Count**	**%**	**Count**	**%**
Issues found	17	8.7	103	52.8	75	38.5
Issues not found	61	67.0	21	23.1	9	9.9

Table 5-4 Importance level for issues found and not found

The classification into 3 quadrants described above can be further refined. Table 5-5 shows the distribution of issues by the assessment type of heuristic violated and the issue quadrant. These figures include all subject issues, not just distinct issues, as is the case above. Many of semantic issues raised by subjects were erroneous. This might indicate that the subjective interpretation necessary in applying semantic heuristic makes them more difficult to identify correctly.

	real-issues found		non-issues found		real-issues not found		total	
	count	**%**	**count**	**%**	**count**	**%**	**count**	**%**
semantic	15	46.9%	12	37.5%	5	15.6%	32	100%
hybrid	89	78.8%	11	9.7%	13	11.5%	113	100%
syntactic	84	48.3%	16	9.2%	74	42.5%	174	100%
total	188	58.9%	39	12.2%	92	28.8%	319	100%

Table 5-5 Heuristic-type by issue quadrant

It may also be the case that where syntactic metrics have been used as proxies for measures of semantic properties, they have been largely misleading. There is also a notably higher proportion of the syntactic issues that were not found (43%) than for the other types of issue. At first sight this seems counter intuitive, as syntactic issues should be amenable to automated metrics and so should be easily found. Further investigation has shown that these unfound issues are mostly trivial, in the fine detail of the design. The subjects generally did not collect suitable metrics to highlight these problems. This large section of trivial syntactic issues can be clearly seen in the Table 5-6, which shows the break down of real-issues by their level of importance. This table also shows that semantic heuristics lead to few, but generally important issues.

	trivial		significant		important		total	
	count	**%**	**count**	**%**	**count**	**%**	**count**	**%**
semantic	2	10.0%	5	25.0%	13	65.0%	20	100%
hybrid	5	4.9%	56	54.9%	41	40.2%	102	100%
syntactic	65	41.1%	63	39.9%	30	19.0%	158	100%
total	72	25.7%	124	44.3%	84	30.0%	280	100%

Table 5-6 Heuristic-type by issue significance

Another way of analysing the issues is by considering the type of entity the heuristics were applied to. Figure 5-9 shows the split of 'real issues' and 'non-issues found'

between the five major types of entity found in an OO system.

It should be noted when reading these charts that the definition of an issue, 'the violation of a particular heuristic by a particular entity', means that there is greater potential for issues relating to certain entity types because there are more of these entities in the system. For instance, since there are 15 classes in the system studied, there is potential for each subject to find 15 violations of each class heuristic. This means that there is potential for far more method and attribute issues than class issues due to the larger numbers of attributes and methods in the system. Similarly the small numbers of system and hierarchy issues reflect their relatively low potential numbers.

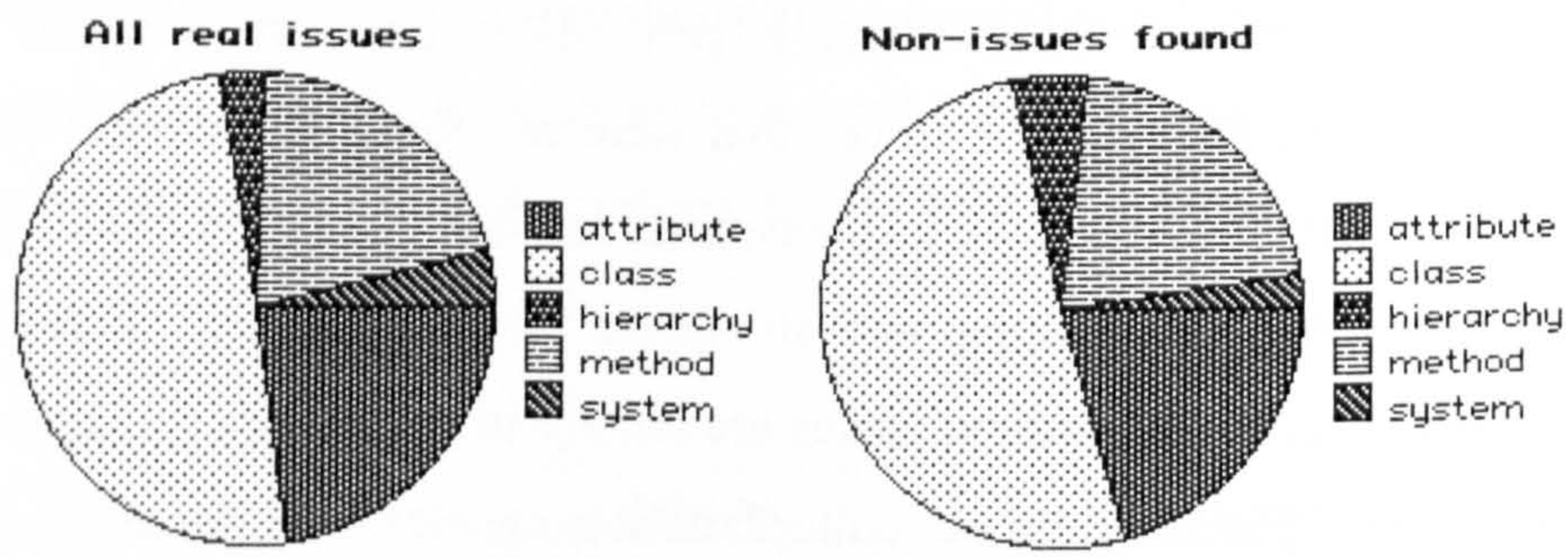


Figure 5-9 Real-issues and non-issues by entity type

Even though there is greater potential for attribute and method issues, around fifty percent of the real issues identified were heuristic violations at the class level. This may suggest that partitioning a system's functionality into a well-designed set of classes is an intrinsically difficult task. Alternatively, it might be that this high level of class issues is an artifact of heuristic maker's preoccupation with the class (but if it isn't problematic why would the class attract this level of attention?)

When comparing the real issues found and the non-issues found, the most striking feature is the remarkable similarity in their distributions. This means that errors were made in proportion to the numbers of real issues that were found for each entity type. It seems the subjects were equally likely to make mistakes whatever the entity type they were looking at.

Figure 5-10 shows the distribution by entity type of issues identified by subjects and

real issues not identified by any of the subjects. The first notable point is the large numbers of attribute issues that went undetected. This is due to the subjects not examining the design in sufficient detail to highlight these issues, i.e., the subjects tended not to use attribute level heuristics. It can be seen that higher levels of system and class level issues were identified relative to the real issues available for the subjects to find. Again this shows that the subjects concentrated on the higher level heuristics.

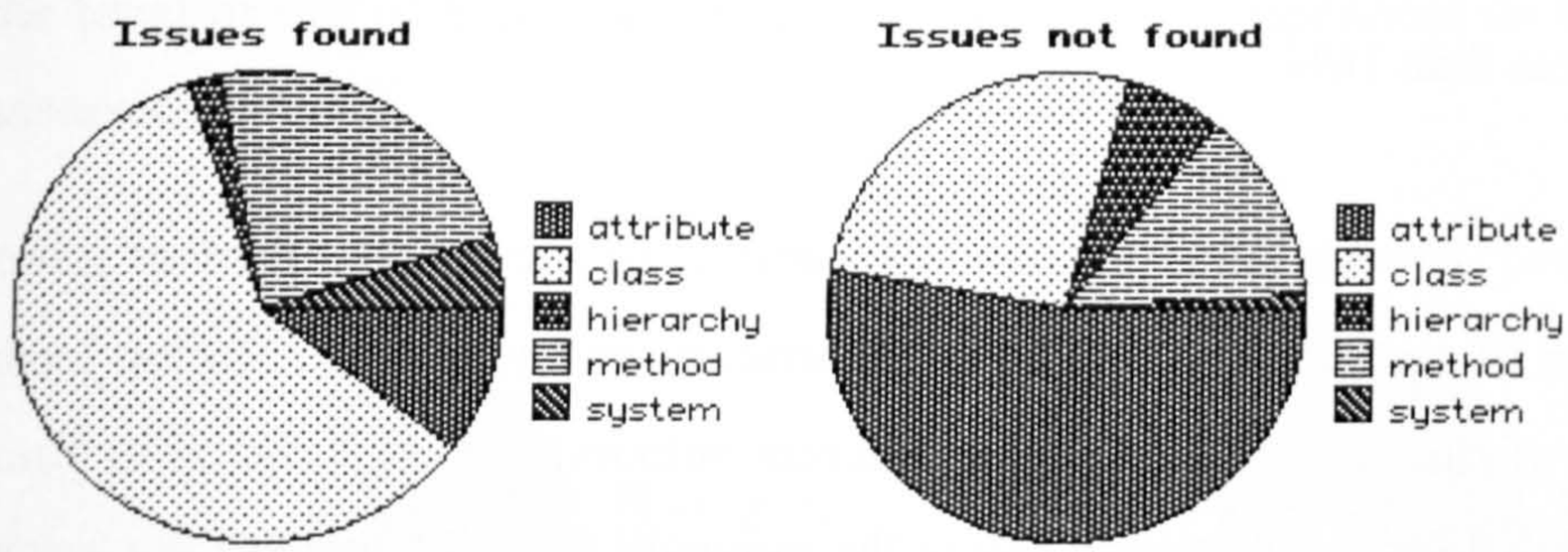


Figure 5-10 Issues found and not found by entity type

5.4.5 Heuristics

5.4.5.1 Analysis of heuristics used

First of all, some summary information on the heuristic-use is presented. There were a total of 56 distinct heuristics used by the 15 subjects and a total of 222 subject-issues found. From figure 5-11 it can be seen that there is a wide variation in the number of issues identified by individual subjects. The subjects found between 4 and 37 issues. Less than half of the subjects found more than 12 issues.

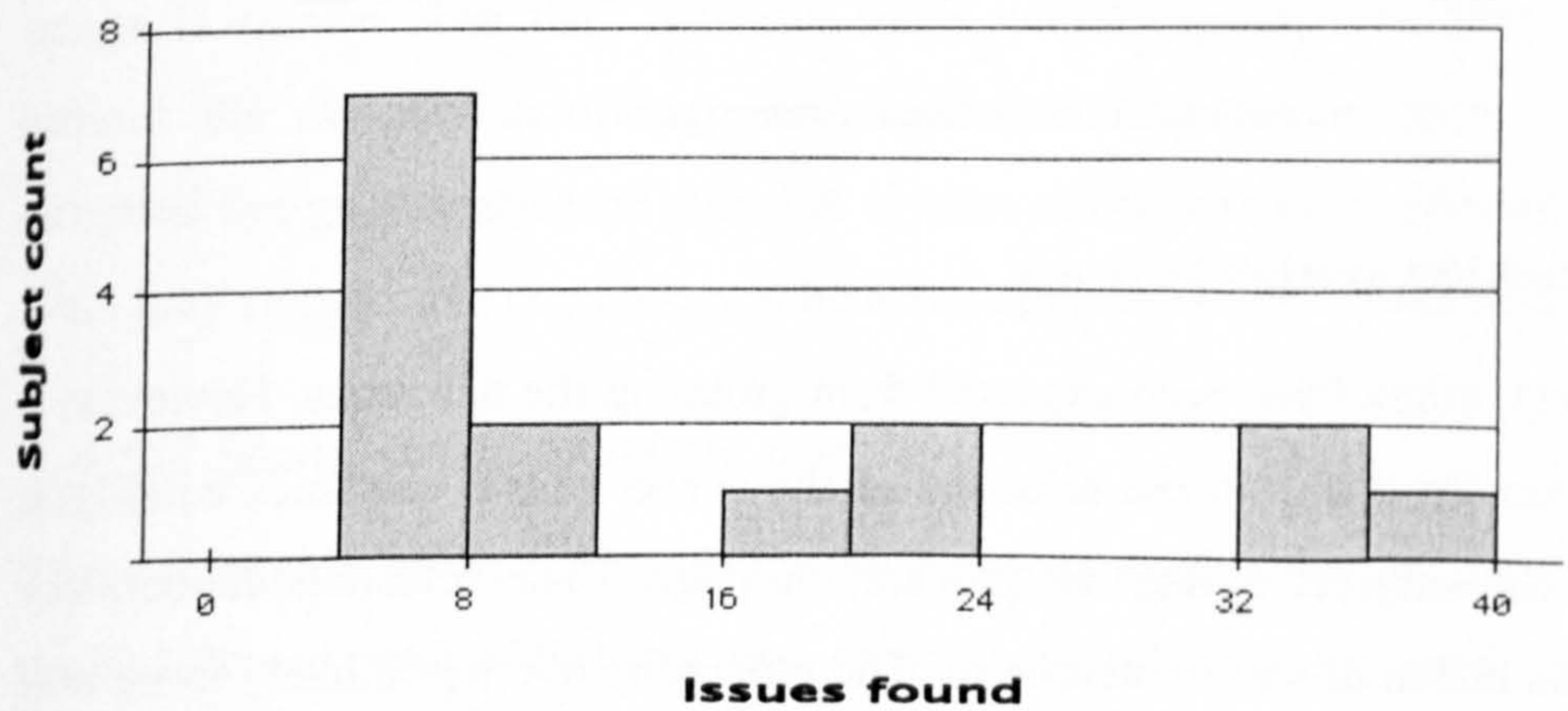


Figure 5-11 Histogram of number of issues found per subject

To further investigate this variation in the behaviour of the subjects, an analysis of the overlap between the issues found by individual subjects was performed. The lighter bars on figure 5-12 show this distribution. It is most likely that a particular issue will only identified by 1 out of the 15 subjects. There is a slightly lower chance of 2 subjects finding the same issue. The chances of 3 or more subjects identifying the same issue are less than 16%.

It might be argued that this small overlap is an artifact of the definition of an issue, i.e., the violation of a particular heuristic on a particular entity. This definition leads to the violation of similar heuristics (from different authors) being treated as distinct issues, and so fails to identify overlap where the rationale for the issue was the same. This effect can be removed by collapsing the set of heuristics into groups of similar heuristics. However when this is done (shown by the dark bars on figure 5-12), only a slight improvement to the overlap between subjects is seen. The modal number of subjects detecting the same issue has only increased from one to two.

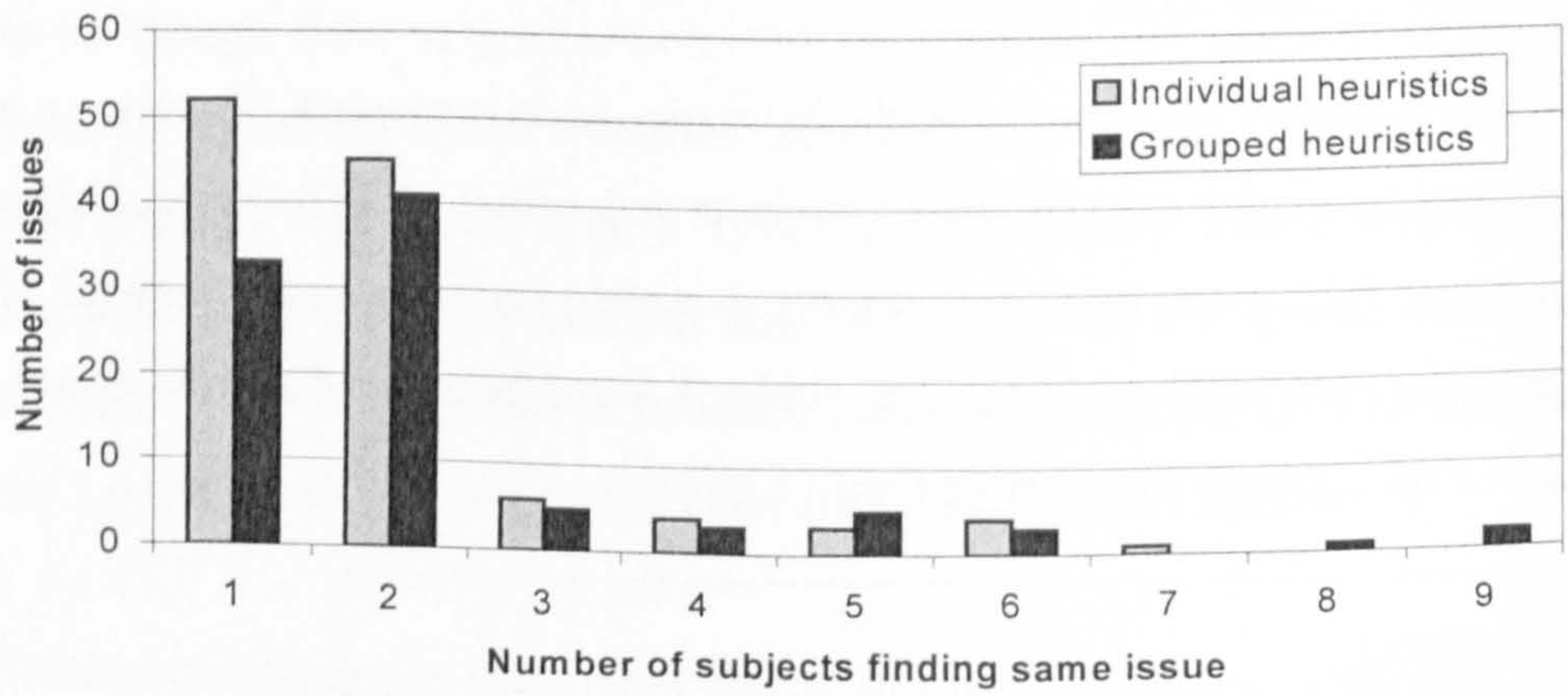


Figure 5-12 Subject overlap on issues

A larger change might have been expected from grouping the heuristics. However, a potential reason for this is in the selection of the subset of the heuristics catalogue presented to the subjects during the preparation stage. The selection deliberately avoided the inclusion of similar heuristics. The remaining overlap is likely to be that between the catalogue subset and the heuristics related to the metrics suites used (and

between those relating to different metric suites).

In a given system, a subject can only use a system heuristic once, but a class heuristic many times (once for each class in the system), and a heuristic for an attribute or method even more times (the two most highly used heuristics apply to attributes and methods respectively). A direct comparison between the usage counts for heuristics of different entity types is, therefore, not very meaningful. It should also be noted that the level of use of particular heuristics is also highly dependent on the system being assessed.

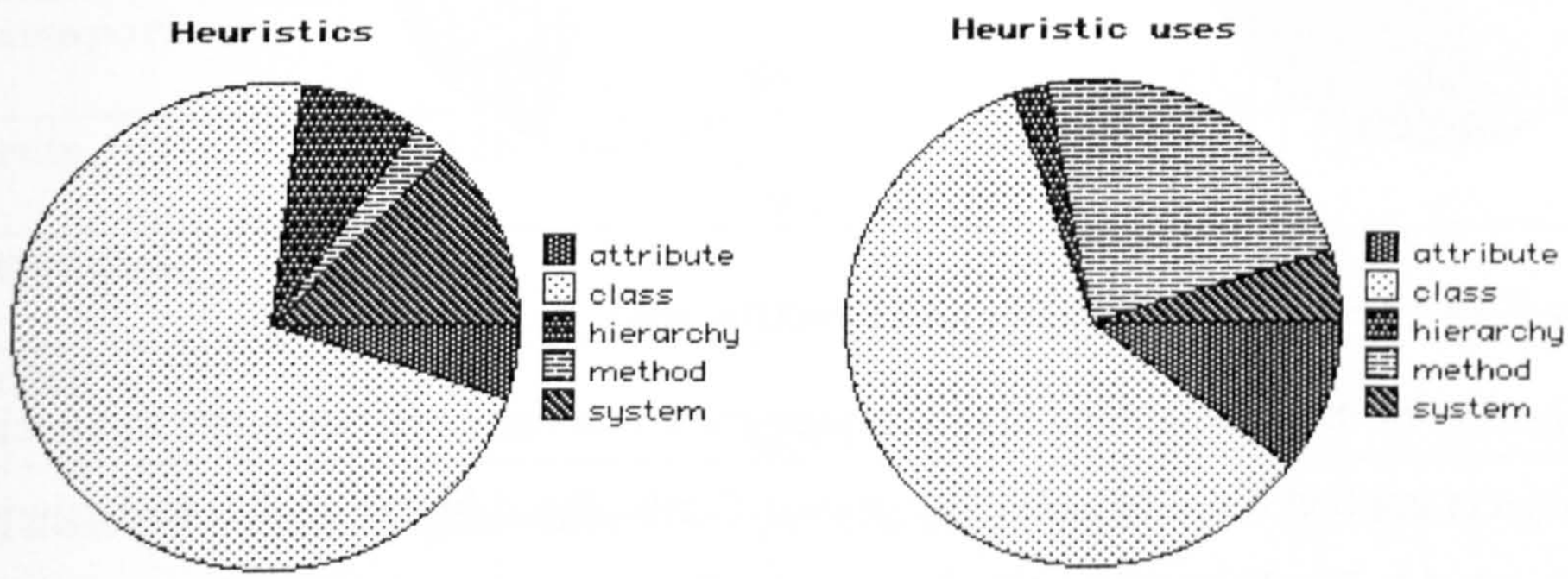


Figure 5-13 Heuristics and heuristic-uses shown by entity type

Comparing the pie charts in figure5-13 also shows this relationship between usage levels and entity-type. The first chart shows the distribution of the 56 heuristics by the type of entity to which they apply. The second chart shows the distribution of heuristic usage by entity type. Method and attribute heuristics have a far larger share of the usage than the heuristic count would otherwise suggest. The class and system usage is smaller than their heuristic count would suggest. The first pie chart also echoes the distribution of suggested heuristics from the literature. Most object-oriented design heuristics are aimed at classes, rather than other entities in the design. This may suggest that other areas of heuristic application have been under-exploited.

5.4.5.2 Stand-alone heuristic use

The description given in the section on heuristic application distinguished between three different types of heuristic use, comparison, rule and threshold. Since the case

study described here is a stand-alone assessment it would be expected that threshold and rule-based heuristics would be used. The charts in Figure 5-14 do show a lower proportion of comparative heuristic *uses* than of comparative heuristics. However, there is still a sizeable use of comparative heuristics in a stand-alone assessment.

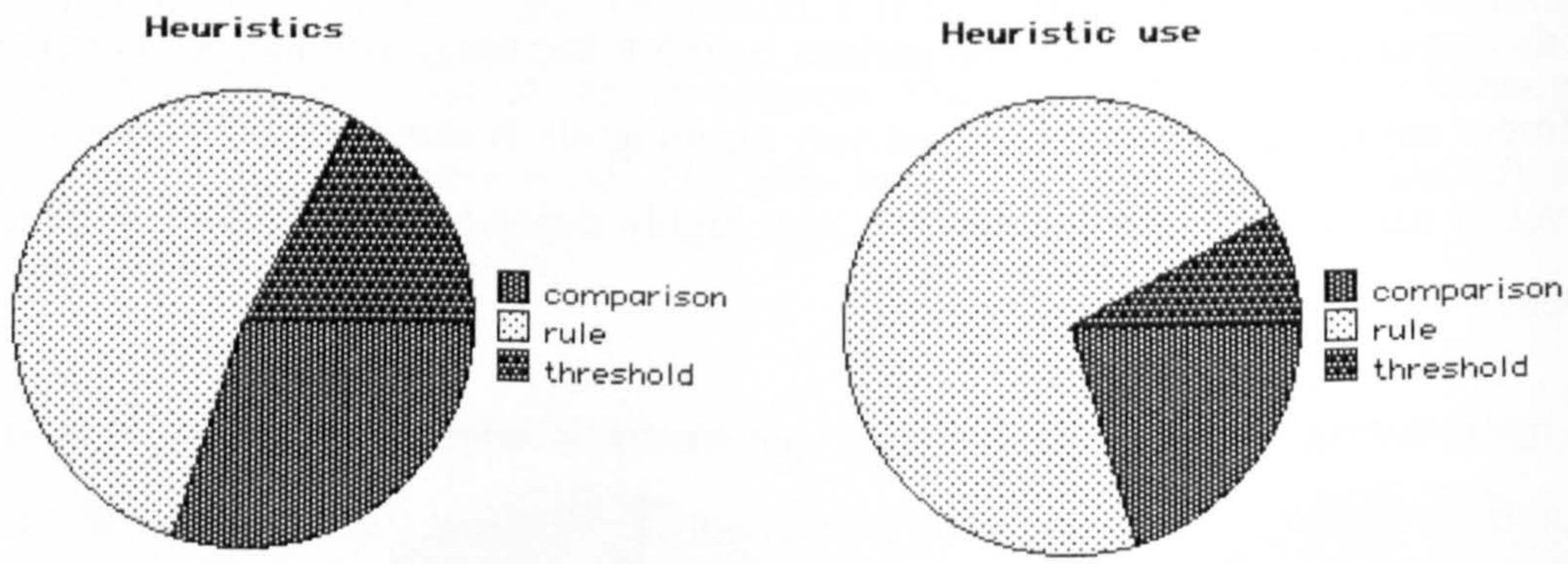


Figure 5-14 Use-type in heuristics and heuristic-uses

Examination of the comparative heuristic uses show that there are two ways in which the subjects applied these to a single system. Firstly, the subjects could create their own thresholds for the comparative heuristics. For example, one subject was using the heuristic 'the higher the inter-object class coupling the more rigorous the testing needs to be' (Chidamber and Kemerer 1994). The subject measured the coupling using the CBO metric and decided that a value of 11 was too high, thus introducing a threshold not included in the original heuristic. The second approach taken was to do within-design comparison. Here an attribute on one entity in the system was compared with other entities of the same type within the system. In this way the subjects effectively identified outliers. For example, Chidamber and Kemerer (1994) state that 'the larger the number of methods that can be invoked from a class, the greater the complexity of the class'. Clearly, a large number of methods that can be invoked are considered worse than a small number, but no particular threshold is given. Rather than introducing their own threshold, some of the subjects noted that a particular class had the largest number that could be invoked (as measured by the RFC metric).

Table 5-7 shows subject performance by use-type. The threshold heuristics, which

should suit this type of assessment, show no false problems. The comparative heuristics also have a fairly low false problem count. The highest error-rate is with the rule-based heuristics. Initially, a relationship between the rule-based heuristics and the error-prone semantic heuristics was suspected for this result. However, the rule-based false problems were found to be almost equally distributed between semantic, hybrid and syntactic. Analysis of the data has yet to provide a satisfactory explanation for the high rate of false problems with rule-based heuristics.

	real problem count	**%**	**false problem count**	**%**	**total count**	**%**
comparison	39	86.7	6	13.3	45	100
rule	126	79.2	33	20.8	159	100
threshold	18	100	0	0	18	100
total	183	82.4	39	17.6	222	100

Table 5-7 Subject performance by heuristic usage-type

5.4.6 Differences between students and industrialists

It has been a point of discussion for some time whether the results of empirical work based on student subjects gave an accurate picture of how industrialists would behave, e.g., (Fenton and Pfleeger 1996; Shepperd 1995). Since this case study contains an almost equal split of students and industrialists, it presents an opportunity to add some data to this discussion.

The data for the number and significance of issues identified appears to show a clear distinction between students and industrialists. The number of issues found by the individual students and the industrialists (summary data shown in Table 5-8) was significantly different (2-Sample t-Test $\alpha = 0.05$). There is also a significant ($\alpha = 0.001$) difference in the distribution of the importance of the issues found between the two groups (Table 5-9).

	Count	Mean	Median	Min	Max
Students	8	21.4	21	6	37
Industrialists	7	8.0	6	4	18

Table 5-8 Summary of issues found by subject type

	Count	Trivial	Significant	Important	Total
Students	8	10	88	43	141
Per subject		1.25	11.0	5.38	21.4
Industrialists	7	1	14	32	47
Per subject		0.14	2.0	4.57	8.0

Table 5-9 Significance of issues found by subject type

Although it is tempting to suggest that these differences are due to the student-industrialist difference between the groups, there are a number of other differences between the groups that might contribute to these results. For instance, the 'industrialists' are generally older (more mature) than the full-time students, their mode of work as part-time students may also have affected the way they performed the task, etc. However the results are interpreted, there was clearly a difference in the way these two groups performed (or at least reported) this task. Consequently, if only full-time students had been used, the results from this case study would clearly have been different.

This is a potentially significant result as much of the research in software engineering is based on work done with student subjects. If there is a suggestion that students and professional engineers behave differently, it brings into question many existing results that have been generalised from student studies to industrial practice.

5.4.7 Subjects' views on measurement

The three pie charts shown in Figure 5-15 represent the answers given in the questionnaire concerning the subjects' views on measurement.

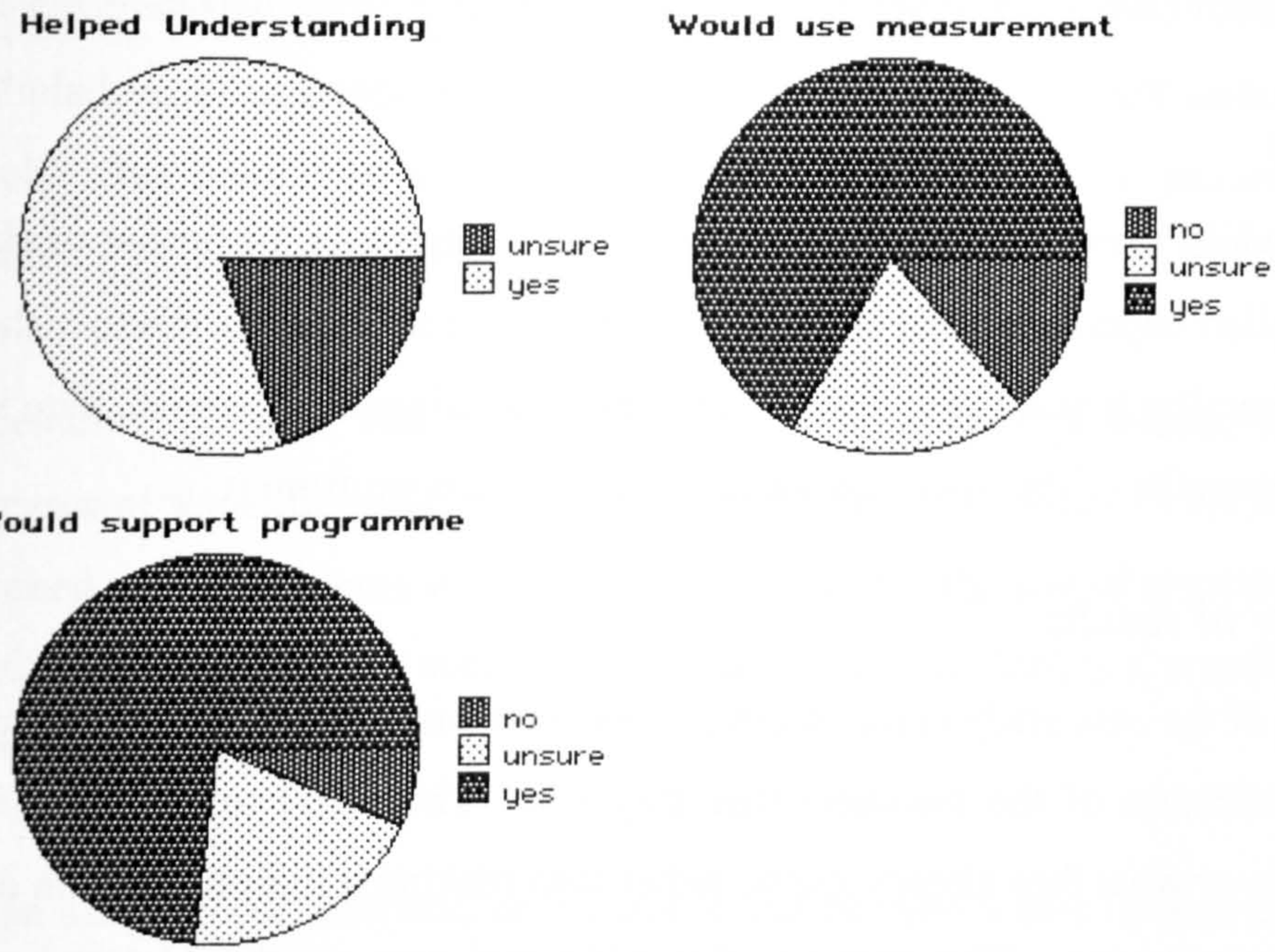


Figure 5-15 Subjects' views on measurement

Question 1: *'Do you feel that relating ideas about quality to measurement data has helped you understand design quality more fully?'*

Most subjects clearly thought that it had. This may indicate that measurement has a role in the teaching of OO design.

Question 2: *'Given suitable tool support, would you choose to apply measurement to help you design future projects?'*

Again, the replies were generally positive.

Question 3: *'Would your experience with this assignment make you more supportive of a suggestion to implement a measurement programme in a company for which you worked?'*

This question considered the change in attitude toward the implementation of a measurement programme following an experience of using measurement. Very few

of the subjects seem to have considered this case study a negative experience.

5.5 Discussion

The case study, whilst providing interesting insights and suggestions for further work, was not a controlled experiment and so no statement can be made as to whether the results would generalise beyond the particular system used. Given this proviso, here is a summary of the results of the work and some conclusions drawn from it.

5.5.1 Summary of results

The first finding of the case study is that users of measurement may not give sufficient thought to the selection of the measures that they use. They tend to use what is provided for them or what they already know, rather than making a logical decision on what they need to measure. This suggests that additional support should be given within measurement tools to aid the user in selecting measures appropriate for their purposes.

In Figure 5-7, the usage profiles for the measures collected by the subjects are given. It was shown how this chart could be used to assess the usage of the individual measures. The level of metric references shows the subject's view of the importance of the measure (or the attribute it purports to measure). The number of times the measure is used to justify a problem shows how useful the measure proved to be (at least within a given system). The ratio of justifications to real justifications shows how accurate the measure is at identifying problems. In the system being studied, coupling and cohesion were considered to be the most important attributes. Of the CK metrics CBO, WMC and RFC were all used to good effect. DIT and NOC were of little use in finding problems with this particular system (there was only one level of inheritance in the system). LCOM was shown to be most likely to lead to erroneous problem identification. These problems with LCOM support the criticisms made by other authors (e.g., (Basili, Briand et al. 1995; Henderson-Sellers, Constantine et al. 1996; Hitz and Montazeri 1996)).

Using syntactic heuristics leads to larger numbers of more accurate issues being

uncovered, but care must be taken to separate the trivial from the important. Violations of semantic heuristics are almost always important issues, however, their subjective nature means that great care must be taken to ensure they are being correctly interpreted and applied.

Genuine semantic issues tend to highlight important problems. However, a high degree of semantic issues raised by subjects were non-issues. This finding is strongly linked to the problems with LCOM. It is felt that the use of syntactic measures (like LCOM) as proxies for semantic attributes (like cohesion) is a practice that should be treated with caution.

The subjects concentrated on measures and heuristics that operate at the class level. This led to a large number of issues at the attribute and method level remaining undetected. However, this was not a significant difficulty, as most of these issues were not considered to be major problems. The relatively large number of heuristics aimed at classes may mean that insufficient effort has been spent identifying heuristics for other entity types. Alternatively, it may highlight the need for additional guidance for designers to help them achieve better coverage of problems in other entity-types.

The subjects' focus on using class-level heuristics echoes OO writer's concentration on writing class-level heuristics. This raises questions about the much-hyped 'intuitive nature' of OO development (Berard 1993; Coad and Yourdon 1991a). If it is so intuitive, why are so many heuristics needed to ensure that the central construct in OO is used properly?

The case study highlighted a wide variation in the behaviour of the subjects. There was a huge range in the number of issues identified and metrics used by subjects, as well as a lack of overlap between the issues identified by individual subjects. It was also shown that users of heuristics would apply comparative heuristics within a stand-alone assessment. They did this either by supplying a threshold themselves when one was not given in a heuristic, or by comparing entities within the design. Within

design comparisons seems a perfectly valid tactic and is effectively a form of outlier analysis. However, the ad hoc creation of thresholds is likely to lead to problems, particularly if there are multiple assessors each working to different values.

Nothing can be said about the general usefulness of particular heuristics from a single case study, but semantic heuristics and rule-based heuristics appear to be more error prone. The problems with the semantic heuristics are due to the use of the LCOM measure. However, the author would suggest that these problems are inherent in the class of measure to which LCOM belongs - those that use of syntactic proxies for semantic attributes. The reason for the poor performance of rule-based heuristics remains unclear.

There were significant differences in way the full-time students and the industrialists carried out the study's task. Since much of the empirical work published within software engineering utilises student subjects, this raises the question of whether the results of such studies can be generalised to industrial practice. The significance of this finding is considered further in the discussion chapter.

The views expressed by the subjects were generally positive, with the large majority of both students and professionals saying that, given suitable support, they would use measurement for future projects.

5.5.2 Threats to validity

It must be acknowledged that this study took place in a university environment as an assignment, and not in the work place on a real project. In general terms, it would be difficult to try to predict whether, and in what way, this may have affected the results. One specific problem is the possibility that the reports and questionnaires reflected what the subjects believed their tutor wanted to hear. To alleviate this problem, it was made clear to the subjects that their questionnaire answers would in no way affect their marks. They were also informed that the analysis of the questionnaires and reports would only begin after the assignment marking had been completed.

Another potential threat is the size of the system used in the case study, 15 classes is not an industrial scale piece of software. However, the task was essentially a design review, and design reviews of this detail would only be done in practice on small subsystems. A complication is that when reviewing a subsystem, the context of the entire system must be considered and this was not present in the case study. As has already been mentioned, the size of the application rendered some of the metrics less useful than might have been the case in a larger application. The example mentioned previously was that DIT was of little use in a system containing only one level of inheritance.

In dealing with a particular application in such depth, some of the results are obviously dependent on the set of potential problems with this application. Some metrics or heuristics may have been used heavily because there were problems with this application that they were useful in highlighting. Similarly, some generally useful metrics may have been underused because the application was not weak in their areas of use.

The subjects used in this study would certainly not be classified as experts and so the results of this study may not be relevant to more experienced practitioners. However, their level of experience is similar to that of many professional software engineers and so there is scope to believe the results have some applicability to industry.

5.5.3 Future work

The work presented here is only part of the analysis from the evaluation section of the case study. There is still work to do on this case study to analyse the changes suggested by the subjects as a result of the initial assessment and of their comparisons between the original version of the system and their altered versions.

There is also evidence that different subjects used different approaches to assessment. This difference might be exploited to improve fault detection in design reviews, by

utilising the reduced issue-overlap between styles in choosing review teams. For this to be possible there would need to be a way of identifying different styles before the review. This might be done by finding some predictor of assessment style or by showing that individuals have consistent styles that can be identified in one review and used in team selection during subsequent reviews.

This work suggests that there may be differences in the way students and industrialists perform software tasks. It would seem prudent to undertake work to directly test the validity of generalising such student-based empirical studies to industry. This is unlikely to prove a straightforward piece of work as the differences between students and professional engineers may be significant in some types of tasks but not in others.

As previously mentioned, there have been far more suggestions in the literature for class level heuristics. This might suggest that more time be spent investigating heuristics for other entity types.

This chapter has concentrated on the application of measurement to design assessment, but this is just one use of design measurement within the software process. Any development activity in which a design is created, altered or reviewed could potentially use design measurement. Empirical studies might also be used to investigate the use of measurement in areas such as initial design formation; design comparison during iterative design refinement (covered in the second part of this case study); measurement within QA and various software maintenance tasks.

5.5.4 Conclusions

The empirical work reported in this chapter was a piece of exploratory work intended to help rectify the lack of empirical support for the utility of measures in product evaluation. In particular, the study involved a design assessment of an object-oriented system. During the design of the case study three main questions were set.

Question1: 'What support is necessary to enable the practical application of design measurement?'

The lack of rationale behind measurement selection shown by the subject suggests that support for measurement selection needs to be supplied. As has been previously mentions there is a general consensus that measurement collection needs to be automated. None of the subjects in this study collected any measures other than those for which there was automated support. It is also clear that anyone who is to use measurement data must be supplied with sufficient information to be able to understand what the measurement means and how it can be used. The subjects avoided collecting measures if they didn't feel they had a good understanding of their purpose. If measures or heuristics are applied without an understanding of their limitations it is likely they will be misapplied.

Question2: 'In practice, how would engineers use measurement results during design evaluation?'

There was a wide variation in the way measures were used by the subjects. The variation was both in the number of metrics and heuristics used and in the approach to using them. Some subjects took a more measurement-centric approach, basing assessment directly on the measurement results. Others took a more heuristic-centric approach using the measurement results as support for assertions about heuristic violations.

The lack of overlap between the issues identified by the subjects has a number of implications. It seems unlikely that similarity of individual reviews could be used as a measure of the reviewer's competence or of the review's completeness. Additionally, it suggests that several reviewers should be used to get reasonable coverage of potential problems.

The subjects didn't always use the measurement results in the manner expected. Their ingenuity in managing to apply measurement results with comparative heuristics during a stand-alone assessment was particularly interesting.

Question3: 'Given the necessary training and experience would engineers choose to apply measurement to help design systems?'

The results have shown that, on the whole, the subjects did a good job of spotting the important high-level problems with the system and it seems the measures were a real aid in doing this. Attitudinal results from the post-task section of the questionnaire suggest that the subjects found the measurement-based approach to assessment helpful and would be more likely to use measurement in design assessment as a result of the experience.

Chapter 6 Discussion

This chapter is divided five sections. Section 1 gives a brief summary of the work done, which provides a reminder of the content and flow of argument from the main body of the thesis. Section 2 summarises the findings of the work and discusses its consequences and significance. An evaluation of the work done for the thesis is presented in section 3. It raises possible weaknesses in the methodology and potential threats to the validity of the empirical results. Section 4 describes how the work could be developed further and what new work has been suggested by this thesis. The chapter concludes in section 5 with a discussion of the main contributions of the work.

6.1 Summary of work done

The goal of this research is to examine the theoretical limitations and practical considerations of people using measurement to assess design quality in real situations. The work focused specifically on the assessment of object-oriented designs. A natural starting point for the investigation was the current state of the art in object-oriented design assessment. The traditional view of quality was presented by discussing validation, verification and evaluation. Previously published methods of software quality modelling were also examined. Within OO design in particular, various design heuristics have been suggested as a means of assessing design quality. The nature of these heuristics was examined and their application discussed. The extent to which the standard qualities identified by quality modelling can be automatically assessed was also examined.

The particular view taken of automation in design assessment was a measurement-centric view. To consider automation of design assessment it is necessary to examine both the extent to which automation is theoretically possible and the mechanisms by which it may be achieved. This means that the mechanisms of software measurement also had to be considered. An introduction to software measurement was presented

which included a definition of what software measurement is and why measurements are taken. Measurement related terminology was also defined to help clarify the discussions. This introduction also discussed the important relationship between measurement and modelling.

If measures are used to assess quality, it is important that the measures correctly represent the attributes they purport to measure. This is the job of measurement validation. Both the theoretical and empirical validation of measures were discussed, including property-base and algebraic techniques. The most widely cited style of validation is based on the application of measurement theory. An extensive discussion of the problems associated with the application of measurement theory is presented.

In considering the manner in which measures are developed and used, it was proposed that the development and the use of measures should be addressed separately. These two aspects of measurement are presented as separate processes showing the stages that measures go through both during their development and their subsequent active use.

One point to emerge from the review of related software measurement research was the need for automation of measurement. A discussion of the facilities available in current measurement tools was presented. This showed the deficiencies in their support for measurement-based design assessment. A new tool was proposed as a proof of concept for some of these facilities. The conception, design, building and testing of this tool were discussed. The completed tool was evaluated against the criteria for which it was built and also against a range of other measurement tools. The evaluation showed that the tool produces accurate measurement results and has a favourable set of features in comparison with both research based and commercial measurement tools.

The empirical section of this research work is intended to investigate the real-world use of automation in the assessment of object-oriented designs. In the study 15

subjects (full-time student and industrialists) were asked to assess the architecture of a small system (15 classes) written in Java using design heuristics and measurement. The results of this evaluation along with answers from a follow-up questionnaire were analysed to show how the subjects went about the task of assessing the design. Evidence was gathered to show the approach the subjects adopted for the task and how effective they were at using automation to assist in the evaluation.

6.2 Summary of findings

This section collects and summarises the findings presented in the body of the thesis. The intention is to show the contribution to knowledge provided by the work. Findings are presented in a chapter by chapter order.

6.2.1 Assessing OO designs

The review of existing quality modelling methods clearly shows the difficulty (and pointlessness) of trying to produce a single definition of quality for all viewpoints, goals and environments. This means that quality models must be built on a per system basis. Anyone reviewing quality must, therefore, be clear about their assessment goals and the viewpoint(s) from which the review is done. Although the quality models themselves may have to be built on a per system basis, it may be possible to create a standard process for deriving these custom models. The author believes that the future quality research should focus in this area.

Two main questions were posed when considering the degree to which quality assessment for OO designs can be automated. The first was to determine the degree to which quality is directly assessable from the information within OO design models. The second was the degree to which this assessment could be automated. To the author's knowledge this was the first attempt to quantify these properties of design quality. The information on what is directly assessable at design-time is useful for anyone performing design reviews or building quality models. Attempts to assess quality factors where the information is not available for direct assessment can lead to dangerous and inappropriate operational definitions of factors. The fact that not all

assessable quality factors can be automated, shows that tools can never produce as complete an assessment of a design's quality as a human reviewer (unless artificial intelligence improves greatly!). It must therefore be accepted that either assessment is partial or that expensive expert effort must be included in any assessment.

The differences in process between 'traditional' and OO development are well documented (de Champeaux 1997; Yourdon 1994). Consideration of this fact in relation to assessment activities shows these differences may introduce additional problems for design assessment. Designs are often left intentionally incomplete compared with what would be possible in the design notation. This means that assessment must be done on incomplete designs. The incremental approach often adopted in OO projects often leads to partially completed, but working, classes that are augmented in later increments. How do you allow for this partial design and construction when applying thresholds and heuristics to such systems? This is still a question for further work.

When considering the automation of design assessment, the use of measurement as the means of automation is a decision that almost makes itself. There are no real alternatives. Once the use of measurement is accepted the question then becomes how are the measurement results to be interpreted? The idea of heuristics as formalised best practice has already been proposed for object-oriented design. Heuristics appeared to be a candidate for interpreting measurement results but additional investigation was necessary into the suitability of heuristics and the relationship between heuristics and measures. This work resulted in the construction of a heuristics catalogue and the discussion of heuristics given in chapter 2.

The heuristics catalogue brings together suggestions for heuristics from a wide range of sources and as such is probably the largest and most complete collection of OO design heuristics available at the moment. The consideration of the properties of heuristics and the ways in which heuristics are applied has led to additional information being stored with the heuristics that should make it simpler to find and apply the correct heuristic for a given situation. This additional information includes

considerations such as trade-offs, contradictions, lifecycle applicability and objectivity. The catalogue should prove useful to those trying to apply OO design heuristics. The discussion on heuristic properties and relationships should give those identifying and publishing heuristics additional guidance on how they should be described (as is done with design patterns).

6.2.2 Software measurement

Attempts to structure the software measurement chapter were initially hampered by difficulties with the exact meaning of widely used terminology. This prompted the attempt to clarify these terms given in the measurement chapter (this was originally started purely for the benefit of the author himself). It is difficult to discuss ideas without a clearly defined and agreed vocabulary. Hopefully this discussion will bring people's attention to the problems with the terminology, and start some discussion leading towards the development of an agreed measurement terminology.

In an attempt to produce a description of software measurement that avoided the problems which initially hampered the writing of the software measurement chapter two innovations were introduced. The first of these was the clear separation of the different elements of measurement. A distinction was made between the techniques available in measurement and the activities in measurement processes. These processes were further distinguished by separating the development of measures from their subsequent use. This was done by describing measurement-related activities within two distinct measurement processes - one for development and one for use. This separation helps to remove any confusion that may occur from the conflation of these distinct sections of measurement. It provides different focuses for the academics developing measures and the practitioners using them.

The second innovation was the central use of modelling in the description software measurement. There have been suggestions in the measurement literature that measurement is meaningless without modelling (Kyburg 1984). An attempt was

made in the thesis to clarify the position of modelling within measurement (and measurement within modelling) and to make modelling central to measurement definition and validation. This is done both by extending the measurement model suggested by Shepperd and Ince (1993) and in describing the activities of measurement in terms of the models built and the activities to validate and apply these models. This view helps to focus on the uses of measurement when defining a measure and clarifies the validation necessary before (and during) real-world use of the measure.

6.2.3 Measurement tool

When deciding on the information from a design which is to be stored in a measurement tool there is a clear trade-off to be considered. It is a trade-off of simplicity and regularity against richness of representation. Richer design information enables more measures to be defined and leads to more flexibility in how designs are analysed. However, it creates difficulties when using the same measures or analysis on design information taken from different sources. Different OO programming languages and design notations share some central concepts, but differ in the detail of the concepts they represent and the manner in which they are represented. This means that central OO concepts considered at a high-level are common to all and are easily stored within a common repository. However, the fine detail of even central OO concepts differs from notation to notation and concepts more on the periphery of OO will be present in some notations but absent from others. This problem becomes apparent when attempts are made to define general OO measures (as observed by Churcher and Shepperd (1995b)) and also in attempts to create cross-source measurement tools. What you gain in specificity you lose in comparability. The implications for measurement definition are that measures which depend on the fine conceptual detail or peripheral concepts will have to have separate definitions for each language or notation to which they will be applied. Measurement tools that implement this type of measure are likely to be those intended for use with a single language or notation.

One design consideration of the tool was how the design information was represented internally. It was observed that the internal format in which the design information is stored would limit the type of measurements that can be taken, and impact how easily particular measures may be implemented. The choice of storage format also imposes the mechanisms for defining measures. An object meta-model and a relational representation were considered for the tool. If the user wants to define measures, a widely used and declarative language such as SQL may be beneficial. If few users actually want to create measures, an object meta-model approach may be more flexible. There is also a tie in with providing guidance on measure selection. Support for a method such as GQM may highlight the need for new measures and increase the likelihood that users will wish to define these measures.

Measurement texts often cite the need for tool support for measurement, e.g., (Grady and Caswell 1987; Fenton and Pfleeger 1996). The testing of the tool provided some hard evidence in support of this. The results produced by the tool were compared with that produced by hand calculation. It was noted that for the system being measured (the 'VocabTutor' from the case study) that the tool took around 2 minutes to run its full set of measures. To hand calculate the same set of measures took 2 days! It has been mentioned previously that assessment cannot be fully automated. If hand collection of the subjective measures is unrealistic in terms of time (and so cost) it may mean that partial automatic collection will be the preferred option. As well as the issue of time there is the issue of accuracy. The hand calculated results contained several errors. Although the task in itself is simple, errors will always occur and it is felt that the tedious repetition involved in measurement calculation was a contributory factor. Tedious and repetitious tasks are always cited as obvious candidates for automation.

Churcher and Shepperd (1995a) presented an argument that counting rules for measures should be more precisely defined. They used metrics from the Chidamber and Kemerer (1994) suite to provide examples to justify their position. Chidamber and Kemerer (1995) replied to this criticism saying that these problems were largely

synthetic and the 'correct' counting rules could be derived by the simple application of common-sense. The comparison of measurement results between MOT and other tools showed substantial differences due to the variation in the counting rules. This provides supporting evidence for the position taken by Churcher and Shepperd (1995a) - that when measures are defined or presented, precise and explicit counting rules should be given.

6.2.4 Case study

Results from both the report analysis and the questionnaire show that users of measurement tend to apply metric suites as provided by their authors without giving more detailed thought to metric selection. This shows that measurement users should be given guidance in selecting what to measure based on their particular goals. Methods such as GQM (Basili, Caldiera et al. 1994) could be usefully employed here. Builders of measurement tools (or CASE tools which include measurement) should consider adding automated support for measurement selection.

Both comments in the subjects' reports and the results from the questionnaires showed that the comprehensibility of measures is important to users. If users don't understand what is being measured or how the results should be applied, they will avoid using a measure. Some measures are simpler or more intuitive than others. Simple, clear measures are likely to be used in preference to complex ones (even if they do not capture the attribute as well). This means that particular care should be taken when describing more complex measures. It should be made clear to the users what is being measured and give examples of how these measures are collected or calculated. This could be aided by measures being initially presented (or later catalogued) in a consistent manner.

The analysis of heuristic use has shown that syntactic heuristics lead to a greater number of issues being found than with hybrid or semantic heuristics. Syntactic heuristics should be capable of complete automation. This means that large numbers of potential problems can be found using automation. However, many of these syntactic issues were trivial problems. This means that care must be taken not to get

bogged-down in the syntactic trivia that an automated assessment tool may generate. It may be worth considering allowing users to ignore trivial issues or concentrate solely on the important issues when using such a tool.

Another result from the case study analysis is that the violation of semantic heuristics usually indicates important problems. Although generalising from a case study should be done with caution, this may have serious implications for automated design assessment. Design quality in OO systems is heavily linked to the semantics of the constructs used (classes, methods, etc.) and these semantic heuristics cannot be automatically checked. This suggests that not spending time on those elements that must be done by hand is likely to result in missing important problems. Clearly there is trade-off here between cost and coverage for managers planning design assessment.

The LCOM measure has previously been criticised by a number of authors and the results from the case study supports these criticisms by showing that the LCOM metric is likely to indicate false problems. The author believes that this tendency to show false problems is due to LCOM being a syntactic proxy for a semantic attribute (cohesion). There is clearly a tendency for metrologists to try to define objective measures for attributes that they perceive to be important, even if this attribute is semantic in nature. The lesson to be learned here must be that those involved with measurement should think carefully before using (or suggesting) measures which create a syntactic proxy for a semantic attribute.

When performing assessment the subjects appeared to concentrate their effort at the class level. A result of this is that problems with other types of construct often went unnoticed. This may be a symptom of poor measurement selection or a natural tendency to look at the class as the main construct for OO implementation. It may be necessary to help assessors to focus on other entity types to ensure a better coverage of possible problems.

The subjects showed a wide variation in approach to assessment. This is shown by the large variation in number of issues identified and metrics used and also by the lack of overlap between the sets of issues identified by different subjects. The low overlap in the issues identified by the subjects suggests that teams (and possibly largish teams) rather than individuals should be used in design reviews. The differences in approach taken by the subjects may be turned to an advantage. If individuals have consistent approaches and these can be identified, it may be helpful in selecting review teams (to optimise coverage and minimise overlap).

When considering the type of heuristics available and how they were likely to be applied, a group of heuristics was identified which codified a tendency. The heuristics said that 'more X was better' or that 'Y should be minimised'. It was considered that this group of heuristics would only be used in comparing alternative entities. However, the subjects unexpectedly applied these heuristics within stand-alone assessment. They did this by creating their own threshold values. Should users be prevented from using comparative heuristics in this fashion? Or, if they want thresholds for the attributes addressed by comparative heuristics, should thresholds be created for them (this would at least ensure consistency). This would in effect be creating new heuristics to fill a perceived need.

Significant differences in behaviour were identified between full-time students and industrialists. This may have major implications for those performing empirical studies in software engineering. Many of the published studies utilise student subjects. This finding raises questions as to whether these results can be generalised to industrial practice. It also questions whether student subjects should continue to be used. However, this result is from a case study and should not itself be generalised to all empirical software engineering research. Further work should be done to support (or challenge) this result.

A final result was that, after the study, the subjects' views were generally positive about the use of measurement for design evaluation. This is an encouraging result, especially for those considering introducing measurement-based assessment. Even if

measurement were actually useful in design assessment it would be hard to introduce without the support of those being asked to use it.

6.3 Evaluation of the research work

It is important with any piece of research to able to dispassionately evaluate the problems and limitations in the work (as well as its strengths). This evaluation will cover the main parts of the thesis in chapter order. The first two chapters discussed, 'assessing object-oriented designs' and 'software measurement', are largely reviews of the associated literature. This means that an evaluation of these chapters must centre on the scope of the literature search, the selection of material and the organisation of the material presented. The chapters entitled 'the design measurement tool' and 'the empirical study' are primarily doing chapters. Here the evaluation is focused more on the work done and method followed.

6.3.1 Assessing object-oriented design

The literature in the areas object-oriented design and software quality is extensive[9]. A problem with writing a chapter of this sort is in the selection of material to include (and omit). The approach taken to selection and presentation was to start by covering the more general questions at a high-level to provide a framework in which to position the more detailed and more closely focussed material towards the end of the chapter. By this approach, the specifically relevant questions of assessability and automatability can be tackled in detail and positioned within the wider body of research without the inclusion of less relevant material from this extensive area of literature.

There may be a tendency in a literature review chapter to simply describe the extant work and not to discuss it with a sufficiently critical eye or provide any new synthesis or analysis from the material. This chapter endeavoured to include a critical analysis of existing quality methods, a cataloguing and structuring of design heuristics, and a new quantitative analysis of assessability and automatability of the standard quality factors.

There were some difficulties encountered whilst writing this chapter. One of these was in attempting to tackle the philosophical aspects of quality. The author found that, although this type of question was an interesting discussion point, it tended not to be helpful in such a practical discipline and metaphysical results are impossible to validate. Another difficulty was in assessing automatability. If a quality model for a system is dependent on viewpoint, goal and environment then the set of 'ilities' to check for automatability will also be dependent. This means that the degree of automatability available will also vary with viewpoint etc. The analysis given of the FCM 'ilities' is therefore only demonstrative and not definitive.

6.3.2 Software measurement

As with the design assessment work, the structuring of a discussion of software measurement in a useful and novel manner is not an easy task. This chapter attempted to avoid a standard presentation by concentrating more on the processes involved in measurement than the measures themselves. It is hoped that this is a more forward looking and useful review than a standard chronological retrospective.

One potential criticism of the thesis is the way that measurement is the only approach to assessment that appears to have been considered. Why only a measurement-centric approach? The answer to this is to be found within the wider context of the thesis, which is concerned with automatability. Automation requires objectivity and no alternative means of objective assessment were found during the literature search.

6.3.3 The design measurement tool

The development of the tool was a somewhat painful and fraught period of the PhD study. There were several difficulties with the development. First of these was trying to building a large and complex tool single-handed in a relatively short period of time. It is often said that such development is required only as proof of concept and needn't be of industrial strength. In this particular case the tool was also to be used in the empirical section of the study. Problems with the function of the tool could adversely impact the conduct of the case study and seriously jeopardise the PhD.

---

[9] The INSPEC database lists 1326 publications in the category of 'object-oriented design' and 4497 publications on 'software quality' (figures taken at 14/06/01)

One of the particular difficulties in the development was its investigative nature. The tool development involved the exploration of what an ideal measurement tool would provide. This meant that the requirements varied substantially during development and meant that the time spent in rework was considerable. Another difficulty was the problem of attempting to produce a cross-source tool. In hindsight, from the point of view of what the tool needed to do for the case study, this was an unnecessary complication, although it had its justification from a more general research viewpoint.

A questionable design decision was the use of a relational representation. While this approach was shown to be plausible, using an object meta-model directly as the representation would have been more flexible. One of the original considerations in making the decision was that with the relational model new measures were created by writing SQL and with the object meta-model new measures would have to be programmed in a high-level language. It was felt that SQL would be more approachable to a larger number of users. However, the number of tool users willing or able to construct their own measures seems very limited[†]. This suggests that the additional flexibility of the object meta-model approach should have been the deciding factor rather than the approachability of SQL.

In general the author would question the wisdom of a PhD in which the success of such a significant piece of development was central the successful completion of the PhD itself. Where the development is in an area that is of interest to commercial software houses there is the possibility of commercial tools overtaking the research development (this happened to an extent to this development). Where development work is undertaken as part of research it shouldn't be too ambitious with the amount of functionality, it shouldn't be too exploratory (should have well-defined fixed specification) and should be used for proof of concept only.

[†] No tool users have contacted the author about creating new measures, despite the measurement tool being available on the web for almost 2 years.

6.3.4 The empirical study

There are standard sets of questions to consider when reviewing the conduct of an empirical study. These are generally grouped into four sections, construct validity, internal validity, external validity and reliability (Yin 1994). Some authors use an expanded fourth category, termed 'conclusion validity', which includes, but augments, reliability (Wohlin, Runeson et al. 2000). In this discussion reliability will be used, as the additional questions posed in conclusion validity are only applicable to formal experiments. Rather than listing all possible threats to validity only those questions which raise potential problems for this particular study will be discussed.

Construct validity considers whether the concepts being studied are correctly represented by the measures being used to describe them. Wohlin lists 'inadequate preoperational explication of constructs' as one of the design threats to construct validity. This means that 'constructs are not sufficiently well defined, before they are translated into measures...' (Wohlin, Runeson et al. 2000). In the case study the subjects were asked to assess the 'quality' of the system being studied, but as has been discussed elsewhere in this thesis, defining quality is not an easy task. The reports produced by the subjects do suggest some variation in their interpretation of 'quality'. Evaluation apprehension is another possible threat to construct validity. Since the task was set as an assignment there is a possible bias due to association of the task with the subjects' course marks (although this was recognised in the design and action taken to try to reduce this effect). As was discussed in the empirical study chapter, it was felt that some of the findings might be dependent on the particular application being studied and this could lead to a lack of generalisability. This problem with using a single system in the study is an example of 'mono-operation bias' (Wohlin, Runeson et al. 2000).

Internal validity deals with establishing the causal relationship between the constructs being studied. The case study described in this thesis is mainly descriptive and exploratory rather than causal so few of the possible threats apply. There are, however, a couple of points worth mentioning. Instrumentation can cause a threat to internal validity. This would include problems of design or wording in data collection

forms (such as the case study's debriefing questionnaire) and also any problems with the measurement tool or training in the operation of the tool. The subject's failure to use the tool in the manner expected must raise questions about the effectiveness of the training they received. Although training was given, and the author believed at the time that the training was sufficient, it may be that more extensive training (or a different approach to training) would have changed the way the subjects used the tool. This in turn may have effected some of the case study's results.

Subjects withdrawing during the course of a study is also a threat to internal validity (this is termed mortality). If withdrawals are not evenly spread in the population this will bias the sample of the population being studied. In the case study there was one subject who, in effect, chose to drop out of the study by deciding not to complete the task in the manner specified. It is however hard to envisage any significant effect on the results from the loss of a single subject.

External validity deals with the degree to which a study's findings can be generalised. The main threats to external validity come from the academic setting of the case study. The system being studied is a relatively small one, the subjects are not all professional engineers and the assessment was done in a university environment as an assignment rather than being performed on a real project.
However Wohlin, Runeson et al. (2000) state that:

> Threats to external validity are reduced by making the experimental environment as realistic as possible. On the other hand, reality is not homogenous. Most important is to characterise and report the characteristics of the environment, such as staff experience, tools, methods in order to evaluate the applicability in a specific context.

Although questions are raised as to whether the results would generalise to industrial practice, there is in reality no standard industrial practice. All that can be done is to report the characteristics of the environment within which the study was carried-out along with the results. This has clearly been done with this study. A further point

often made regarding generalisation from case studies is that no attempt should be made to generalise the results themselves. In a case study the results should only be generalised to theory which then must be tested by subsequent experiment.

Reliability concerns the repeatability of the operations and measurements within a study. There are a number of threats to the repeatability of the study. The use of subjective 'expert' judgements, e.g., the categorisation of issue importance, is a potential problem. More than one evaluator was used to make these judgements more reliable, but some variation must still be expected if this was repeated by different evaluators. There were also some fuzzy areas of interpretation when transcribing the information from the subjects' reports into the case study database.

In hindsight there were a number of problems with the case study that might have been avoided. The task that was used produced a large and complex structure of data, which is difficult to extract and analyse. Assessing the quality of a design is a very wide ranging task and refining the task to reduce its scope would probably have helped the data complexity problem. The author would also recommend not underestimating the time necessary for data acquisition and analysis for such complex data sets. Another problem with data collection concerned the action logging within measurement tool. In the design of the case study it was intended that action logging in the tool would be another source of data. The subjects did not use the tool in the manner expected and this left little useful data in the logs. It might be useful to try to get more interactive use of the tool by the subjects to facilitate better triangulation of evidence through action logging. Forcing the subjects to give more thought to metric selection by removing the predefined metric sets from the tool might do this. An approach that might be taken in the future would be to further develop the tool to give more assistance with metric selection and result interpretation. A preliminary study may have helped to identify some of the problems with the final study. However, as has already been mentioned, the case study was fixed in time by the teaching calendar and it proved impossible to schedule a preliminary study in the time available.

Case studies are sometimes considered to be a second-class citizen of empirical

studies. There is a feeling that research should be done with the utmost rigour and that only formal experimentation has this necessary rigour. As Yin (1994) states 'investigators who do case studies are regarded as having insufficient precision, objectivity, and rigor'. However, there are several forms of empirical approach available to the researcher, each with their own strengths and weaknesses, and a predilection for experimentation may result in the most appropriate technique not being used. In the design of the empirical work the author's initial inclination was a preference for experimentation. Practical and ethical considerations meant that this was not possible and so a case study approach was used. This proved to be fortuitous. The use of a case study approach rather than an experiment allowed a much richer view of the subjects' use of measurement in assessment. This was felt to be particularly beneficial in view of the limited amount of previous work in this area. With little prior knowledge of how subjects would use measurement it would have been extremely difficult to design a good experiment. This means that 'chance' practical considerations actually forced a correct methodological decision.

6.4 Further work

The findings of a piece of research are not its only outcomes. Another type of outcome is the questions that it raises and the opportunities for new work that it presents. This section reviews suggested further development of the work done toward this thesis and discusses new areas of work that this research makes possible.

The work on heuristics presented in the 'assessing object-oriented design' chapter can clearly be taken further. More work is necessary in refining the structure of the heuristics catalogue and analysing the trade-offs and contradictions between the individual heuristics. Additional work is also needed to continue the collection and discovery of heuristics. An analysis of the heuristic catalogue (and the heuristics used in the case study) shows a large number of class level heuristics. This may indicate that many heuristics for other entity-types that have not been found (due to concentration on classes). Alternatively, there may simply be more design considerations with classes and hence more heuristics defined for them.

The review of the measurement literature shows that there are a number of areas where further work is necessary. These include:

1. deciding what is needed to fully define a measure;
2. moving towards agreement on what is considered sufficient validation for measures (sufficient before publication and sufficient before real-world use);
3. agreeing standard definitions for measurement terminology and ensuring the interaction between the various dimensions of measurement description are fully understood.

Further development could clearly be made to extend the facilities of the design measurement tool. Obvious further work includes:

1. creating input mappings for addition sources of data (e.g., other OO languages and CASE tools);
2. adding one or more 'MeasurementMethods' to the tool to give assistance in measurement selection;
3. adding assistance in result interpretation, e.g., thresholds and acceptable ranges, outlier identification and design comparisons;
4. extending the number of measures available in the tool

The results presented from the empirical study are an investigation of the use of measures in 'stand-alone' design assessment. There are many other development activities in which measures might usefully be applied. The first step in this direction will be the analysis of the second half of the case study data, which deals with the use of measurement to compare design alternatives.

It has been mentioned previously that case studies can only be generalised to theory. This clearly leaves additional work to be done in designing and running formal experiments to try to confirm the relationships suggested by the study. One of the more important results for this type of further investigation is the seemingly different behaviour between student subjects and industrialists. If this difference were confirmed it would have major implications for the design of empirical studies in

software engineering. Such studies are often performed with student subjects but the study's designers often intend their results to generalise to industry.

The apparent differences in the subjects' behaviour could be further investigated. As mention earlier, these differences of approach could be applied to help maximise coverage and minimise overlap in design reviews. Further investigation would need to establish whether distinct types of assessment approach do in fact exist. There would also have to be additional work to check whether real benefit could be gained from using this reviewer classification in team selection.

6.5 Conclusions

Rather than making a single large advance, this thesis makes a number of smaller but still significant steps on a number of fronts. The quantitative analysis of assessability and automatability of quality factors for object-oriented designs indicates the degree to which OO design assessment can theoretically be automated. Measurement was proposed as a mechanism for automation. Consideration of current measurement techniques led to a new model-based structure for considering the process of software measurement. The extended view of object-oriented design heuristics provides a mechanism for linking measurement and assessment. The development of a flexible measurement tool provided many insights into the functionality that measurement collection tools should supply and the potential problems in their development and deployment.

A new empirical study was described which assessed how these techniques work together in practice to facilitate the automated assessment of object-oriented designs. The study has shown that measurement can successfully be used to aid design assessment. However, automation can introduce its own problems. Care must be taken to avoid getting bogged down in the syntactic trivia that a tool can generate. The desire for automation can also lead to the definition of syntactic measures for semantic attributes, such measures may be problematic. The study also shows the importance of semantic issues in design assessment and in doing so shows the

continued need for human input.

This work examines a particular use of software measurement - object-oriented design assessment. There is a body of existing work that uses measurement to assess quality by looking for correlations between measures and external quality indicators. This thesis is distinctive because it examines a use of measurement that aids designers in finding specific problems at design time. It also shows how people actually use measures in this practical situation.

Appendices

Blank

In

Original

Appendix A OOD heuristics catalogue

The heuristics collected below are generally given verbatim, but some have been reworded to present ideas as positive heuristics or to enable the ideas to be presented outside of their original context.

A.1 Heuristic classification

A large number of general design principles for OO systems have been proposed. These heuristics are given below. They are collected into groups relating to the same OO construct/idea. Some heuristics relate to more than one group, e.g., heuristics for coupling in inheritance hierarchies. In these cases, a rather arbitrary decision must be made as the group within which a particular heuristic is presented.

The break down of the groupings is as follows:

- Inheritance
    - Depth of hierarchy
    - When to use / not use inheritance
    - What inheritance should model
    - Multiple inheritance
    - Miscellaneous inheritance
- Aggregation
- Association
- Cluster
- Class
- Abstract Classes
- Coupling
- Cohesion
- Information Hiding
- Message Passing

- Service
- Attribute
- Protocol
- Polymorphism
- Dynamic Modelling
- Miscellaneous

As well as grouping heuristics by construct, the heuristics may be classified in a number of other ways.

Heuristics may give guidance on a **product**, i.e., the design itself. They may be guidance for the **process** used for producing the design or the **notation** used to express the design. They might also be heuristics for **managers** trying to control the process. This classification is shown in the second column of the heuristic tables, '**Prod / Proc / Not / Man**'.

A further way of subdividing these heuristics is by the point in the **life cycle** at which they may be applied. The following abbreviations are used:

Life-cycle Activities

AN	:	Analysis
AD	:	Architectural Design
DD	:	Detailed Design
CO	:	Coding
ALL	:	Applies across all life-cycle activities.

AD - DD Means a heuristic is applicable from architectural design to detailed design.

Another way of classifying heuristics is by their likely method of assessment. Heuristics may be objective, partially objective, or subjective. Objective heuristics can

be objectively assessed and quantified. Objective/subjective heuristics can be measured to some extent, but subjective assessment of the result of this measurement is still required. Subjective heuristics are wholly dependent on the subjective appraisal of the designer or reviewer.

Obj - Objective assessment

Sub - Subjective assessment

Obj/Sub - Partially objective

Some of the heuristics may also be dependent upon the method being used for the design, or the language intended for its implementation. The final column relates these dependencies.

Meth - Method dependent

Lang - Implementation language dependent

A.2 Inheritance

A.2.1 Depth of hierarchy

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment Style	meth/lang dependent
Systems of around 100 classes should have a depth of hierarchy of approximately 7 ± 2 levels. Excessively deep hierarchies may be difficult to modify. (Coad 1991c)	PROD	AN-CO	Obj / Sub	lang
Avoid inheritance structures that are too shallow (3) or too deep (7). (Firesmith 1995)	PROD	AN-CO	Obj / Sub	lang
Flat structures and hierarchy depths >5 are less maintainable than a depth of 3. (Daly, Brooks et al. 1996)	PROD	AN-CO	Obj / Sub	lang
Inheritance hierarchies should be deep. (Riel 1994)	PROD	AN-CO	Sub	
Inheritance hierarchies shouldn't be too deep (more than six levels) or designers will get lost in the hierarchy. (Riel 1994)	PROD	AN-CO	Obj / Sub	lang

The deeper in the hierarchy designers get the more confusing things are. Conceptual entropy also increases in an evolving application hierarchy. (Dvorak 1994)	PROD	AN-CO	Obj / Sub	

Table A-1 Depth of inheritance hierarchy heuristics

The ideal depth of hierarchy seems to be based on a trade-off between reuse and understandability. Reuse encourages deeper hierarchies to make the best use of commonalties within a system. Excessively deep hierarchies are difficult to maintain because they are difficult to understand. There is also a suggestion that flat systems are also more difficult to maintain (Daly, Brooks et al. 1996). A possible reason for this is that for most OOPLs polymorphism only works within the hierarchy of implementation inheritance. It is generally held that systems can use polymorphism to increase maintainability. If a system has a flat structure it can have no polymorphism and so may be less maintainable.

At one extreme, there is a lack of maintainability and reuse because the hierarchy is too flat. At the other extreme, there is a lack of maintainability due to difficulty of understanding because the hierarchy is too deep. Exactly where in the middle ground we consider the ideal depth will depend upon the particular problem under consideration, and our prioritisation between reuse and maintainability.

The ranges given for 'ideal' depth of hierarchy are 5-9 Coad (1991c), 3-7 Firesmith (1995), 1-5 Daly, Brooks et al. (1996). Coad's heuristic of 5-9 was suggested for systems of around 100 classes. Daly's suggestion of 1-5 was based on an academic example of around 20 classes. Firesmith doesn't specify the size of system his heuristic referred to. The limited data we have, of a recommended depth of 5-9 for 100 class systems, and 1-5 for a system of around 20 classes, would indicate that the best depth of hierarchy for a system is also dependent on the size of the system.

Even if a relationship were found which would give an 'ideal' depth of inheritance for a particular system size this would only be a guideline. Hierarchies shouldn't be forced to have a certain depth of hierarchy or to sit within a particular range if this

would be inappropriate for the system in question. However a depth of hierarchy outside of the usual range would merit a re-evaluation of the design to verify that the hierarchy had been well structured.

Those of the heuristics that give specific numbers for depth of inheritance may be considered language dependent. This is because the different way the hierarchies are organised and structured may cause some languages to typically have deeper hierarchies than others.

A.2.2 When to use inheritance

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment Style	meth/lang dependent
Inheritance should only be used for subtyping (as opposed to implementation inheritance) (Bar-David 1992)	PROD	AD-CO	Obj / Sub	
Subclasses should always be subtypes. (Leavens 1991)	PROD	AD-CO	Obj / Sub	
Only use inheritance when you want to inherit all the properties of a subclass, not just some of them. (Rumbaugh 1993)	PROD / PROC	AD-CO	Obj	
The assertions of subclasses should be consistent extensions of the assertions of their parents. (Firesmith 1995)	PROD / PROC	AD-CO	Obj / Sub	
The use of inheritance for code reuse without subtyping should be avoided. (Rumbaugh 1993)	PROD / PROC	AD-CO	Obj / Sub	
Replace implementation inheritance with aggregation except in the case of mixins. (Firesmith 1995)	PROD / PROC	AD-CO	Obj / Sub	
Any subclassing mechanism for reuse should produce 'clean objects'. (Lee and Tsai 1993)	PROC	AD-CO	Obj / Sub	meth lang
Don't confuse a class with the classes needed to implement it. Just because a class is implemented using other classes shouldn't mean it inherits from them. (Rumbaugh 1993)	PROD / PROC	AD-CO	Obj / Sub	
Subclasses should not delete features of their superclasses. (Firesmith 1995)	PROD	AD-CO	Obj	

Subclasses that overwrite more than 3 inherited operations may not be specialisations of their superclasses. (Firesmith 1995)	PROD	AN-CO	Obj / Sub	
A class which doesn't use or rejects many of the attributes and services of its superclass is generally a poor specialisation. (Coad and Yourdon 1991b)	PROD	AD-CO	Sub	
Use inheritance when the resulting subclasses will have different features and behaviours; otherwise use instances of power classes. (Firesmith 1995)	PROD / PROC	AD-CO	Obj / Sub	
Subclassing may be specified in terms of incremental differences, or the specification of some restriction (constraint) over and above those imposed on the base class. (Rumbaugh 1993)	PROD / PROC	AD-CO	Obj / Sub	lang meth
Subclasses should be different from their superclasses. (Firesmith 1995)	PROD	AN-CO	Obj / Sub	
Do not model the dynamic semantic of a system as an inheritance hierarchy. (Riel 1994)	PROD	AN-CO	Obj / Sub	
Avoid derived classes which attempt to cancel out operations defined for the base class. These derived classes are not part of the specialisation hierarchy. (Riel 1994)	PROD	AD-CO	Obj / Sub	
Never create a new class if you can reuse an existing one. (Firesmith 1995)	PROD / PROC	AD-CO	Obj / Sub	
All the methods in the specialisation interface should be appropriate for specialisations of the given object. (Berard 1993)	PROD	AN-CO	Sub	
There shouldn't be any application specific methods in the specialisation interface. (Berard 1993)	PROD	AD-CO	Obj / Sub	
A subdimension should be a specialisation of the dimension it partitions. (McGregor and Korson 1993)	PROD	AN-AD	Obj / Sub	

Table A-2 When to use inheritance

There seems a strong movement in the literature away from implementation inheritance without substitutability and toward inheritance only for defining subtypes. There are several reason put forward for this shift of emphasis in the literature.

Implementation inheritance may give high levels of reuse, but is achieved through complex procedures involving method deletion, redefinition and, restriction. These convoluted techniques lead to messy, difficult to understand code, which in turn causes problems for the testers and maintainers of this code.

Another problem is that in most OOPLs inheritance may be used for either specialisation, subtyping or subclassing, and it is difficult to identify in the final code which of these was originally intended. Using inheritance to model what are largely orthogonal concepts within a single hierarchy (or set of hierarchies) in this way confuses the reasoning underlying the design of the hierarchy. OOPLs which separate the subclass and subtype hierarchies side-step this problem to some extent, but confusion may still exist when some classes in these hierarchies are true specialisations of their base classes and others are not.

A further problem often cited is that changes in a base class may introduce unwanted methods into subclasses. This is only true if the inheritance mechanism used specifies which methods are not to be inherited and would not be true for those inheritance mechanisms which explicitly specified those which are to be inherited.

There are any number of definitions of subtyping given in the literature, without producing a formal definition they may be summarised as follows. The 'uses protocol' for a subtype should be a superset of the union of the protocols of its supertypes. Here the 'uses protocol' is defined as a description of what is available at public interface. This includes method names, the typing of method arguments, and return values, as well as the definition of the behaviour of each of the methods in the interface under any given situation. An alternative way of putting this is that a subtype should be both type substitutable and behaviourally substitutable with all of their supertypes in all contexts. All of the other heuristics aimed at ensuring substitutability will be observed if these definitions are observed.

Where code reuse without subtyping is desired, it has been suggested that aggregation

should be used instead of inheritance (Rumbaugh 1993).

Lee's heuristic about clean object subclassing isn't generally supported by either methods or languages (Lee and Tsai 1993). Although Rumbaugh's statement that specialisation can be either by incremental differences or by restriction is strictly true, specialisation by restriction is not generally supported in methods or languages.

A.2.3 What inheritance should model

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment Style	meth/lang dependent
An inheritance tree should be a generalisation-specialisation tree with more specialised capabilities near the leaves of the tree. (Love 1991)	PROD / PROC	AN-CO	Sub	
You will have little trouble with incorrect use of inheritance if an instance of the subclass is in every respect an instance of the superclass. (Rumbaugh 1993)	PROD	AN-CO	Sub	
Specialisation inheritance is better than interface inheritance, which is better than implementation inheritance. (Firesmith 1995)	PROD / PROC	AD-CO	Obj / Sub	
Use specialisation inheritance to capture recognisable 'a kind of' taxonomies. (Firesmith 1995)	PROD / PROC	AN-AD	Sub	
Inheritance should be used to model gen/spec relationships. (Korson and McGregor 1992)	PROD / PROC	AD-CO	Obj / Sub	
Inheritance hierarchies should be specialisation hierarchies. (Riel 1994)	PROD/ PROC	AN-CO	Sub	

Table A-3 What inheritance should model

There is a clear agreement here that inheritance should be used to model specialisation hierarchies. This point taken together with the recommendation from the previous section means that inheritance should be used for type substitutable specialisations. Since this usually must be done through the language mechanism for code reuse Rumbaugh's statement that the subclass should be in every respect an instance of the superclass seems to summarise the perceived best practice concisely.

A.2.4 Multiple inheritance

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment Style	meth/lang dependent
A class being defined via multiple inheritance should have at most one subclass in a given dimension. (McGregor and Korson 1993)	PROD	AN-AD	Obj / Sub	lang
Only use multiple inheritance for: • Specialisation inheritance • Adding capabilities using mixins. • For subclasses whose protocol is divided into separate roles. • For subclasses that are combining superclasses from two dimensions. • To separate interfaces from implementations. (Firesmith 1995)	PROD / PROC	AN-CO	Obj / Sub	lang
A subclass should never have more than two direct superclasses. (Firesmith 1995)	PROD	AN-CO	Obj	lang
A mixin should be combined with a single generalisation to produce a specialisation with one or more focused capabilities. (Firesmith 1995)	PROD / PROC	AD-CO	Obj	lang
Try to reduce multiple inheritance to a single inheritance hierarchy by asking the question "are any of my base classes subclasses of one another?" (Riel 1994)	PROD/P ROC	AN-CO	Sub	lang
Watch out for the implementation of containment hierarchies as multiple inheritance. (Riel 1994)	PROD	AN-CO	Obj / Sub	lang
Common base classes in multiple inheritance typically require virtual inheritance to model the problem. (Riel 1994)	PROD	DD-CO	Sub	lang
When adding multiple inheritance to a OOD assume a mistake is being made and prove otherwise. (Riel 1994)	PROD	AN-CO	Sub	lang
A subclass being derived via multiple inheritance should inherit from at most one class in a dimension. (Firesmith 1995)	PROD	AN-CO	Obj / Sub	lang

Table A-4 Multiple inheritance heuristics

According to Firesmith (Firesmith 1995) multiple inheritance should only be used for:

- Specialisation inheritance.
- Adding capabilities using mixins.
- For subclasses whose protocol is divided into separate roles.
- For subclasses that are combining superclasses from two different dimensions.
- To separate interfaces from implementations, especially within reuse repositories.

Firesmith gives no clear indication of how multiple inheritance could be used to separate interfaces from implementations (point 5). Surely you would use an abstract class to specify the interface and use single inheritance to produce different implementations based on this interface?

Firesmith also states that a subclass should never have more than two direct superclasses. This however could cause problems in the following circumstances:

A class which requires two additional capabilities added through mixins.
A class which requires the capability provided by a mixin but is also derived from classes from two dimensions.
A class specialises from two parent classes but also requires a mixin.

These examples show that it is easy to produce subclasses which might require three or more direct superclasses. To produce these classes and still use no more than two direct base classes would require additional levels to be added into the inheritance hierarchy. This produces a trade-off between: increasing depth of hierarchy, which causes difficulty in understanding the overall structure of the hierarchy; and additional complexity, and difficulty in understanding, of the directly inherited composition of the individual class.

McGegor & Korson (McGregor and Korson 1993) and Firesmith (Firesmith 1995) agree that a class should have at most one parent class in a particular dimension. The dimensions of an entity can be defined as a set of orthogonal classifications which

apply to that entity. If all the possible classes in a dimension form a disjoint partitioning of the dimension, as stipulated by McGregor and Korson, then this heuristic seems self evident. Classes not obeying this rule could not be true specialisations of all of their superclasses.

All of the heuristics in this section have been marked as language dependent because not all languages support multiple inheritance (most notably Smalltalk).

A.2.5 Miscellaneous inheritance

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
Use generics rather than inheritance where appropriate. (Firesmith 1995)	PROD / PROC	AD-CO	Obj / Sub	meth
Creating and assessing the protocol hierarchy and the conformance diagram can show some problems with class hierarchies. (Cook 1992)	PROD	AD-CO	Obj / Sub	
Avoid mixing different levels of abstraction within the same model. (Rumbaugh 1993)	PROD	AN-AD	Sub	
Subclasses within a dimension should be defined so that any instance of the parent class clearly belongs to exactly one subclass. (McGregor and Korson 1993)	PROD / PROC	AN-AD	Obj / Sub	
The portion of a class' specification that is specialised to form the subclass within a dimension should be disjoint from the parts of the class' specification used to form other dimensions. (McGregor and Korson 1993)	PROD	AD-DD	Obj / Sub	
Try to extend hierarchies by adding new classes at the leaf nodes, not near the root. (Firesmith 1995)	PROD / PROC	AD-CO	Obj / Sub	
Superclasses should standardise their subclasses. (Firesmith 1995)	PROD / PROC	AD-CO	Obj / Sub	
Each instance of a subclass should also indirectly be an instance of its superclass. (Firesmith 1995)	PROD	AD-CO	Obj / Sub	
Use specialisation inheritance to permit the polymorphic substitution of subclasses for superclasses. (Firesmith 1995)	PROD / PROC	AD-CO	Obj / Sub	

The extents of subclasses in a dimension should partition the extent of the superclass. (Firesmith 1995)	PROD	AN-CO	Obj / Sub	
Keep classes in inheritance structures portable and interoperable. (Firesmith 1995)	PROD	CO	Sub	
Make the hierarchy's conceptual architecture explicit. (Dvorak 1994)	PROD / PROC	AN-CO	Sub	

Table A-5 Miscellaneous inheritance heuristics

Rumbaugh (Rumbaugh 1993) recommends avoiding the mixing of different levels of abstraction within the same model. This recommendation lacks a clear way of deciding whether classes are at different levels of abstraction. Are there indeed clear and distinct levels of abstraction? If so what are they? If it is accepted that there are not clear and distinct levels of abstraction then there must be a continuum of levels of abstraction. Whether classes are at sufficiently different levels for it to be considered a design fault then becomes subjective.

Firesmith recommends trying to extend hierarchies by adding new classes at the leaf nodes, not near the root. This has the obvious short term advantage of reducing the extent of the changes to the hierarchy. However, if classes added to a hierarchy are forced into positions in the hierarchy in which they do not really belong, in an attempt to obey this rule, the longer term negative implications for the maintenance of the hierarchy are likely to out weigh the short term advantages. The overriding consideration in adding to a class hierarchy must be the overall conceptual integrity of the hierarchy itself, not any short term savings whilst making the additions.

Firesmith states that specialisation inheritance should be used to permit the polymorphic substitution of subclasses for superclasses. Although substitutability though subtyping doesn't in theory require inheritance, in practice most OOPLs only support polymorphism and subtyping through implementation inheritance. As a practical heuristic this is therefore perfectly valid.

McGregor and Korson state that a sub-dimension should be a specialisation of the

dimension it partitions. Firesmith makes the same point in a slightly different way saying that the extents of subclasses in a dimension should partition the extent of the superclass. A further suggestion of McGegor and Korson is that the portion of a class' specification that is specialised to form the subclasses within a dimension should be disjoint from the parts of the class' specification used to form other dimensions. Since dimensions represent orthogonal concepts it is reasonable to assume that the sections of a class relating to different dimensions will have little interaction. This does however create problems with class cohesion.

The first heuristic in this section shows that the use of generics is marked as method dependent. Although most major OOPLs support generics in some form (Smalltalk being the notable exception), direct representation in the methods is generally lacking.

A.3 Aggregation

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
Aggregation should not be used to represent non-compositional relationships, i.e., spatial/temporal inclusion, attribution, class membership, attachment, ownership. (Civello 1993)	PROD	AN-AD	Obj / Sub	
Methods should allow for more subtle representation of aggregations. They should support: visibility, separability, immutability, sharing, ownership. (Civello 1993)	PROC / NOT	AN-AD	Obj	meth
Designers should be clear about the precise relationship intended (see above) for each use of aggregation in a design. (Civello 1993)	PROD / PROC	AN-AD	Obj / Sub	
Containing classes should know what they contain, but contained classes should have no knowledge of who contains them. (Riel 1994)	PROD	AD-CO	Obj	
If a contained element has behaviour then at least some of that behaviour should be used by the containing object. (Riel 1994)	PROD	AN-CO	Obj / Sub	
Distribute system intelligence down the containment hierarchies. (Riel 1994)	PROD	AD-CO	Sub	

Keep containment hierarchies narrow and deep. (Riel 1994)	PROD	AN-CO	Sub	
Do not make containment hierarchies too deep (no more than six levels) or designers will get lost in their own hierarchy. (Riel 1994)	PROD	AD-CO	Obj / Sub	

Table A-6 Aggregation heuristics

Although Civello indicates that aggregation should not be used for attributes this does often occur. Both attribution and aggregation are usually implemented as class member variables in OO languages. This lack of conceptual distinction in the language often leads to aggregation being used for attributes in design.

The other two heuristics from Civello are related to the overly simplified treatment of aggregation in most methods. It would be desirable for such issues as, visibility, separability, immutability, sharing and ownership to be addressed in the design. This would allow for a more complete description of an aggregation, to the level at which the implementation of aggregation can be fully specified and verified. Ideally these considerations will be included in the method. If they are not, the designer should still consider them to gain a clear idea of how each aggregation would best be implemented.

Riel's first three heuristics are related to each other. To be considered as a separate class to be aggregated and not an attribute of the containing class, the contained class must have its own behaviour. Distributing intelligence down the hierarchy means that the contained class should be given as much behaviour and responsibility as possible. In this situation containing classes must know about the classes they contain in order to request these services. However, aggregation can also facilitate reuse, if the contained class makes no reference to its containing class the contained class can be reused without also having to reuse the containing class.

The question of depth of hierarchy for aggregation appears to be similar to that for inheritance. Too great a depth and the hierarchy becomes difficult understand. Too

shallow a hierarchy and less than optimum use is made of reuse and information hiding. Although there is only one suggested maximum depth given here, it seems reasonable to assert that 'ideal' depth of aggregation hierarchy as for inheritance will depend upon the size of the system. The similar natures of the trade-off in depth in both cases further suggests that aggregation hierarchy depth should normally be approximately the same as inheritance hierarchy depth for systems of the same size.

A.4 Association

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
When given the choice of using association or containment, chose containment. (Riel 1994)	PROD	AD-CO	Obj / Sub	
Implement one-to-one and many-to-one unconditional associations as references. Implement all other associations with association classes. (Riel 1994)	PROD	DD-CO	Obj	lang
Both classes in an association have equal status. If one class is subordinate to the other then aggregation or attribution should be used. (Rumbaugh 1996)	PROD	AN-AD	Obj / Sub	

Table A-7 Association heuristics

The first heuristic suggests using containment rather than association where possible. This is really saying that we should use the strongest form of relationship which is applicable in a given situation. It was described in the section concerning aggregation that such considerations as visibility and mutability affect the nature (or strength) of a relationship and this holds equally well for association as it does for aggregation. With this in mind the second heuristic is a little over simplistic and the ideal mapping from design to implementation for associations and aggregations will be far more involved than this, and may differ from language to language.

A.5 Cluster

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent

Classes should be grouped into defined collections of upto twenty classes that have restricted visibility from other classes in the system. (Love 1991)	PROD / PROC	AD-CO	Obj	lang meth
A "system of objects" should represent an object-oriented concept, as opposed to a functional concept. (Berard 1993)	PROD	AD-CO	Obj / Sub	
With a system of objects, all of the component objects should directly support the object-oriented concept that the system represents. (Berard 1993)	PROD	AD-CO	Obj / Sub	
If a system of objects presents multiple interfaces to the outside world, each of the interfaces should represent a complete and coherent object-oriented concept. (Berard 1993)	PROD	AD-CO	Obj / Sub	lang meth

Table A-8 Cluster heuristics

The idea of clusters (or subjects or ensembles) is to provide a higher level grouping/information hiding construct. A first consideration is how large these groups should be? Love suggests upto 20 classes, while Coad goes for the ubiquitous 7 ± 2
(Coad and Yourdon 1991a). While this type of heuristic helps guard against extremes the real question is how we decide what to put in the clusters.

Berard states that a cluster should represent a clear object-oriented concept, rather than a functional one. He also states that all components should directly support this object-oriented concept.

Support for higher level structural class groupings and multiple interfaces is limited in both methods and languages.

A.6 Class

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
Avoid having too many services per class. Each class typically has no more than six or seven public services. (Coad 1991c)	PROD	AD-DD	Obj / Sub	

Given some desirable behaviour in a system, a decision must be made as to the class in which to place it. The following criteria have been offered for consideration when making such a decision. • Reusability- would this behaviour be useful in more than one context. • Complexity- how difficult is it to implement the behaviour. • Applicability- how relevant is the behaviour to the type in which it might be placed. • Implementation knowledge- does the behaviour's implementation depend upon the inter details of a type. (Booch 1991)	PROD	AD	Obj / Sub	
Avoid excessive attributes. On average only one or two attributes are needed per service. (Coad 1991c)	PROD	AD-DD	Obj / Sub	
Avoid 'fuzzy' class definitions. A class should be cohesive with a single, well defined and clearly bounded purpose. (Coad 1991c)	PROD	AD	Sub	
A class should have no more than six data members. (Riel 1994)	PROD	AD-CO	Obj	
Keep related information in one place (class). (Riel 1994)	PROD	AN-CO	Sub	
A class which has no operations should not be a class (it is probably an attribute). (Riel 1994)	PROD	AN-CO	Obj	
If a class has only one or two operations then it should not be a class (it should probably be implemented as an operation on some other class). (Riel 1994)	PROD	AD-CO	Obj / Sub	
Semantic constraints on a class shouldn't be implemented in the class structure if this leads to a proliferation of classes. (Riel 1994)	PROD	AD-CO	Obj / Sub	
Beware classes which have only one instantiation in a system. (Riel 1994)	PROD	AD-CO	Obj / Sub	
A class should have easily identifiable and nameable behaviours. (Coplien 1992)	PROD	AN-CO	Sub	

A class should not represent too large or too small an abstraction. (Coplien 1992)	PROD	AN-CO	Sub	
Pre-ordained H/W components and their S/W interfaces are good candidates for system entities. (Coplien 1992)	PROD	AN-AD	Obj / Sub	
If you can't think of two good significantly different implementations, then a broader abstraction may be needed. (Coplien 1992)	PROD	AD-DD	Obj / Sub	
Each class should attempt to do its own validation. (Rising 1994)	PROD	DD-CO	Obj / Sub	
Beware of classes that are really attributes. (Rising 1994)	PROD	AN-CO	Sub	
Beware of classes that are really operations. (Rising 1994)	PROD	AN-CO	Sub	
The number of encapsulated methods in the public interface for any given object should be kept to a manageable number. (Berard 1993)	PROD	AD-CO	Sub	

Table A-9 Class heuristics

Several of the heuristics relate to the numbers of services and attributes that a class should contain. Berard suggests that the number of methods in the public interface of a class should be kept to a manageable number. Coad is a little more specific and suggest 6 or 7 as the maximum. There are also considerations of the minimum number of services. Classes with no services would usually be better represented as an attribute in another class. Classes with only one or two services might be better shown as operations in another class.

Riel suggests that classes should have no more than 6 data members. Coad relates the number of attributes to the number of services saying that only one or two attributes are needed per service. Again there are minimum considerations for number of attributes. Classes with no attributes are likely to be the result of functional rather than object-oriented decomposition and are usually better represented as services in other classes.

Given some desirable behaviour and some information needed to perform this behaviour, a decision must be made as to the class in which to place each element of behaviour and each piece of information. Classes should be the repository for related

information and behaviour.

Coplien suggests that if two good significantly different implementations can't be found for a class then a broader abstraction may be needed. This embodies the idea that classes should hide design decisions. The interface of the class should not depend upon its implementation, in this way the implementation may be changed without affecting any other classes with use services provided by this interface.

A.7 Abstract classes

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
Abstract classes should capture common abstractions. (Firesmith 1995)	PROD	AN-AD	Sub	
Use deferred classes to control future subclassing if appropriate. (Firesmith 1995)	PROD	AD-CO	Obj / Sub	
A class should not constrain the development of its future subclasses. (Firesmith 1995)	PROD / PROC	AD-CO	Obj / Sub	
Abstract classes should probably not have concrete ancestors. (Firesmith 1995)	PROD / PROC	AD-CO	Sub	
Abstract superclass rule (ASR) states: all superclasses must be abstract. (Hursch 1994)	PROD / PROC	AN-CO	Sub	
Base classes should be abstract to facilitate the maintenance of a specialisation hierarchy. (Riel 1994)	PROD/P ROC	AN-CO	Obj / Sub	

Table A-10 Abstract class heuristics

Firesmith (Firesmith 1995) states that abstract classes should capture common abstractions. There can be little argument about this. There would be little point in producing an abstract class unless it captures commonalties between two or more existing classes (or envisaged future classes).

Two further heuristics given by Firesmith appear to clash somewhat. Firstly he says

that deferred classes should be used to control future subclassing if appropriate, and also that a class should not constrain the development of its future subclasses. Unless a class constrains the development of its future subclasses how can it control future subclassing? The important point here must be that classes should not unnecessarily constrain the implementation of future subclasses. Nor should they constrain the interfaces of future subclasses without clear purpose.

Another heuristic suggested by Firesmith is that an abstract class should probably not have concrete ancestors. The situation of a abstract class having concrete ancestors is a fairly unusual one, so you might say 'an abstract class *will* probably not have concrete ancestors.' However, there is no apparent reason why they *should* not.

A.8 Coupling

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
Avoid having a large, central object which controls most of the functionality. (Riel 1994)	PROD	AD-CO	Obj / Sub	
Minimise the knowledge one class has of others, so when one class is changed, it is less likely to change the others. (Berard 1993)	PROD	AD-CO	Obj / Sub	
The more system components that make use of a particular object, usually the better is the design of the overall system. (Berard 1993)	PROD	AD-CO	Obj / Sub	
The larger the number of other distinct system components with which an object must deal directly, usually the poorer is the design of the overall system. (Berard 1993)	PROD	AD-CO	Obj / Sub	
Ensure that relationships among components of the system are as tenuous as possible. (Berard 1993)	PROD	AD-CO	Sub	
The coupling of objects should take place only on an application by application basis. (Berard 1993)	PROD	AD-CO	Obj / Sub	meth lang
Classes near the root of an inheritance tree should not depend upon classes further down the tree. (Love 1991)	PROD	AN-CO	Obj / Sub	
A base class should not have derived class information. (Rising 1994)	PROD	AD-CO	Obj	

Coupling- 'on one hand, weakly coupled classes are desirable; on the other hand, inheritance - which tightly couples superclasses and their subclasses - helps us to exploit the commonality among abstractions. (Booch 1991)	PROD	AN-CO	Sub	
Classes should have the minimum of coupling with other classes. (Love 1991)	PROD	AD-CO	Obj / Sub	
Eliminate unnecessary relationships among modules. (Myers 1978)	PROD	AD-CO	Sub	
Minimise the tightness of those relationships that are necessary. (Myers 1978)	PROD	AD-CO	Obj / Sub	

Table A-11 Coupling heuristics

Most of the standard coupling heuristics given above are straight forward and well known. However Berard makes the seemingly curious statement that 'the more system components that make use of a particular object, usually the better is the design of the overall system.' Normally this type of interconnection would be included in any measurement of coupling and would therefore give a high coupling value considered to be an indication of poor design. The significant point here is the separation of the relationships 'uses' and 'is used by'. For one class to use another requires that the first class has knowledge of the class it is using. This means that it is difficult to modify the class being used without modifying the class that is using it. It also means that the *using* class cannot easily be reused without the *used* class also being reused. From the point of view of the class being used things are quite different. It has no need of any knowledge about who is using it, and so does suffer the problems which result from this. This style of 'uni-directional' coupling was also suggested by Sharble & Cohen who used this type of definition of coupling in metrics calculation (Sharble and Cohen 1993).

Separate consideration should be given to coupling in relation to inheritance. Booch states that there is again a trade-off when considering the coupling in relation to inheritance, "On one hand, weakly coupled classes are desirable; on the other hand, inheritance - which tightly couples superclasses and their subclasses helps us to exploit the commonality among abstractions" (Booch 1991). Such heuristics as there

are concerning inheritance and coupling try to reduce the coupling between the subclass and superclass whilst still exploiting their commonalties. This is done by imposing restrictions on the way subclasses and superclasses access each others elements. Love (Love 1991) and Firesmith (Firesmith 1995) agree that superclasses should not depend upon, or have visibility of, their subclasses. This means that any changes made in a hierarchy will only propagate down toward the leaf nodes and not back up toward the root.

Love also suggests that objects should not directly access data defined in their superclasses. This allows the implementation details of the superclass to be changed without affecting its subclasses. This does have a slight performance overhead because of the need for additional function calls. For most systems this slight reduction in performance is not significant, this heuristic should therefore be applied for all but the most performance critical systems.

A.9 Cohesion

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
Ensure all the member variables and methods of a class make sense as part of the class. (Berard 1993)	PROD	AD-DD	Sub	
Make sure that subsystems are comprised of parts that are all highly logically related. (Berard 1993)	PROD	AD-DD	Sub	
Any one module should represent a single, coherent concept, at a uniform level of abstraction. (Berard 1993)	PROD	AN-CO	Sub	
A software engineer should be able to understand the responsibility of a module without understanding the module's internal design. (Parnas, Clements et al. 1983)	PROD/PROC	AD-CO	Sub	
Each of the methods in the public interface for an object should perform a single coherent function. (Berard 1993)	PROD	AD-CO	Sub	
An object removed from the context of the immediate application, should, in isolation, still represent a coherent and complete object-oriented concept. (Berard 1993)	PROD	AD-CO	Sub	

An object containing only functions is not cohesive in an object-oriented sense. (Berard 1993)	PROD	AD-CO	Obj	
An object containing only data is not cohesive in an object-oriented sense. (Berard 1993)	PROD	AD-CO	Obj	
Well-designed objects should contain only primitive methods in there public interface, and their should be at least a sufficient set of primitive methods - and preferably a well-thought-out complete set of methods. (Berard 1993)	PROD	AD-CO	Obj / Sub	
All encapsulated methods should be application-independent. (Berard 1993)	PROD	AD-CO	Obj / Sub	meth lang
A classes services should related to each other and to the class in an intuitively satisfying way. (Coplien 1992)	PROD	AN-CO	Sub	
A change to one part of a system should have minimum impact on other parts. (Coad 1991a)	PROD / PROC	AD-CO	Obj / Sub	

Table A-12 Cohesion heuristics

Most of the heuristics relating cohesion are rather fuzzy and ill defined. Their general basis is that the services and attributes of a class are logically related to each other. The class itself should be based around a single well defined abstraction and should remain a coherent whole irrespective of the context in which it is used. Services should follow the same cohesion rules as apply to non-OO functions, and should be application independent.

A.10 Information hiding

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
A change to a class should require the minimum number of design decisions in other classes to be reconsidered. (Coplien 1992)	PROD	AD-CO	Obj / Sub	
Variables should only be accessed or modified through the sending of messages to, or the invoking of, the accessing methods. (Wirfs-Brock and Wilkerson 1989)	PROD / PROC	DD-CO	Obj	

An accessing method should only store or retrieve the value of its associated variable. It should perform no other computation. (Wirfs-Brock and Wilkerson 1989)	PROD	DD-CO	Obj	
Follow a policy of strict data hiding. (Riel 1994)	PROD	DD-CO	Obj	
A type should be a logical black box specification of externally visible behaviour. (Rumbaugh 1993)	PROD / PROC	AD-CO	Obj / Sub	
It should be possible to change the implementation of one module without knowledge of the implementation of other modules and without affecting the behaviour of other modules. (Parnas, Clements et al. 1983)	PROD	AD-CO	Obj / Sub	
Do not use protected data, it weakens data hiding. Use protected access functions instead. All data in the base class should be private. (Riel 1994)	PROD / PROC	AD-CO	Obj	lang
Superclasses should not have visibility of their subclasses. (Firesmith 1995)	PROD / PROC	AD-CO	Obj / Sub	
Allow access to inherited state information only via methods. (Berard 1993)	PROD / PROC	AD-CO	Obj	
Objects should not directly access data defined in their superclasses. (Love 1991)	PROD	AN-CO	Obj	

Table A-13 Information hiding heuristics

Most of the information hiding heuristics given here represent universally agreed points: data should only be accessible via access functions; types should represent black box specifications independent of internal implementation (hiding design decisions). Wirfs-Brock's statement that accessing methods should perform no other function than the storage or retrieval of its associated value is open to question. Does this mean that data validation cannot take place before storage? Can audit information about data accesses not be maintained during accesses?

An on-going argument regarding information hiding is that of C++'s *friends* and similar constructs. Friend classes are allowed access to private data which breaks strict information hiding rules. However access for friends is given on a class by class basis, which means that minimum visibility is given to the accessed data. The

alternative would be to provide access functions for the data concerned. This would maintain information hiding but allow indirect access to a larger set of classes, which need not have visibility of this data.

Some interesting points occur when information hiding and inheritance come together. The conventional wisdom and usage has been to consider a subclass as a single entity including all of the data and functions inherited from the superclass and directly defined in the subclass. Conceptually this is the case, as an instance of the subclass represents a single object. However, in practice, implementation considerations must also be taken into account. One of the reasons behind information hiding is to restrict the extent to which the change of one class will require changes to others. A subclass has been traditionally allowed direct access to the data (and private functions) of its superclass. This means that changes to the internal implementation of the superclass, which would not affect classes that use the superclass, may well necessitate changing subclasses.

The heuristics given above involving inheritance are suggestions made to mitigate this effect by removing direct access to the superclass' data. Rather than treating a subclass and the superclass data and functions it inherits as a single object, the superclass part of a subclass object is treated as a separate object from which the subclass object requests services. The services that may be requested are those in the specialisation interface as well as the standard external interface.

A.11 Message passing

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
Message passing may be used to : 1. request a corresponding service 2. provide notification of an event 3. provide data (actual parameters). (Firesmith 1993)	PROD / PROC / NOT	AN-CO	Obj / Sub	meth lang
Minimise object collaboration. In most cases objects should only collaborate with three to five other objects. (Coad 1991c)	PROD	AN-AD	Obj / Sub	

Minimise collaborators to 6 for a given system requirement. The overall number may be higher. (Rising 1994)	PROD	AD-CO	Obj / Sub	

Table A-14 Message passing heuristics

Message passing in OO is considered conceptually a different activity from function calling in the procedural paradigm. This can be seen from Firesmith's three uses of message passing. Numbers one and three used together map to a procedure call. The main difference is event notification. This need not be point to point, as a procedure call must be, but may be broadcast. The object giving notification of an event need not (in principle) know how the receiving objects will react to the notification. A procedure call will have a return value whilst event notification is one way.

In principle we have this distinction between message passing in the object-oriented paradigm and function calls in the procedural paradigms. In practice, however, OOPLs are run on procedural machines with stack frames and return addresses. This limits the scope of usage of message passing. Running OO code on a virtual machine organised around OO constructs can solve the problem. Software systems like CORBA which handle message passing at a higher level, and even Windows to an extent, with it's event queuing system, allow a more natural OO use of message passing.

A.12 Service

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
Usually if a service requires more than three parameters something is wrong. (Coad 1991c)	PROD	AD-CO	Obj / Sub	
OO services are quite small, typically five to ten source statements. (Coad 1991c)	PROD	DD-CO	Obj / Sub	
Services should do some work, not just pass the buck to another service. (Love 1991)	PROD	DD-CO	Obj / Sub	
Services with 100+ statements, or with deeply nested blocks, or with extended CASE statements, should be examined to see if they can be restructured. (Coad 1991a)	PROD	DD-CO	Obj / Sub	

OO code should have a minimum number of branching statements. No more than one branching statement for every five lines of code. (Love 1991)	PROD	DD-CO	Obj / Sub	
Services of the same name in different classes should mean the same things. (Love 1991)	PROD	AD-DD	Sub	
Services of different names should mean different things. (Love 1991)	PROD	AD-DD	Sub	
Services should obey the same coupling and cohesion rules as apply to functions in structured methods.	PROD	DD-CO	Obj / Sub	
Keep protocols as simple as possible, have no more than 6 data parameters per method. (Rising 1994)	PROD	AD-CO	Obj	
Any override must return the same value as the base class function would when applied to the base part of the derived class object. (Bar-David 1992)	PROD	DD-CO	Obj	lang
Derived class override must have the same signature as the overridden base class service. (Bar-David 1992)	PROD	AD-CO	Obj	lang
An operation on a class should not have the potential for changing the semantics of behaviour of the class. (Coplien 1992)	PROD	AD-CO	Obj / Sub	
The number of lines of source code for well-designed encapsulated methods are usually very small. (Berard 1993)	PROD	CO	Sub	

Table A-15 Service heuristics

Services are largely the same as procedures and should obey the same coupling and cohesion heuristics. Services are usually smaller than procedures and excessively large or complex services should be questioned. There is usually less branching in services because branching based on type should be replaced with polymorphic service requests.

A.13 Attribute

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent

If a characteristic affects behaviour in the domain, use inheritance or containment, otherwise, use an attribute. (Rising 1994)	PROD	AD-CO	Obj / Sub	
If derived classes have no meaningful behaviour, attributes should have been used. (Rising 1994)	PROD	AD-CO	Obj / Sub	
Attributes should contain useful information about the objects of a class, but should not have associated behaviour. (Rising 1994)	PROD	AD-CO	Obj	
Attributes should hold encapsulated, unshared values. (Rumbaugh 1996)	PROD	AN-CO	Obj	

Table A-16 Attribute heuristics

This again summarises to the difference between attribution and aggregation. If behaviour is associated with data then aggregation should be used. If the value of an attribute effects the behaviour of a class then inheritance should be used.

A.14 Protocol

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
Keep protocol as simple as possible. (Riel 1994)	PROD	AD-CO	Sub	
Implement a minimal public interface which all classes understand. (Riel 1994)	PROD	AD-CO	Obj / Sub	
Do not put implementation details such as common-code private functions in the public interface. (Riel 1994)	PROD	DD-CO	Obj / Sub	lang
The public interface for an item should be consistent with the overall concept embodied by the item. (Berard 1993)	PROD	AD-CO	Sub	
The protocol of a subclass should be a superset of the union of the protocols of its parents. (Firesmith 1995)	PROD / PROC	AD-CO	Obj / Sub	

Table A-17 Protocol heuristics

What is meant by protocol is generally the full extent of what is available at the interface. This includes method names, parameter and return types, preconditions and postconditions.

A.15 Polymorphism

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
It is usually preferable to replace CASE and IF-THEN-ELSE statements with an appropriate gen-spec structure and polymorphic service calls. (Coad 1991c)	PROD / PROC	AD-CO	Obj / Sub	
To localise the effect of changes and hence reduce maintenance effort the use of polymorphism is essential. (Haythorn 1994)	PROD	AD-CO	Obj / Sub	
Great classes are essentially interface specifications embodied as abstract classes. We don't create the abstract class because it is evident in the domain - we create it to control the impact of future changes in the domain. (Haythorn 1994)	PROD / PROC	AD	Sub	
Undisciplined use of polymorphism causes similar problems to the use of gotos. (Ponder and Bush 1994)	PROD / PROC	AN-CO	Sub	
Use polymorphism rather than switch/case statements for decisions based on type. (Firesmith 1995)	PROD	DD-CO	Obj / Sub	

Table A-18 Polymorphism heuristics

Ponder's (Ponder and Bush 1994) position is somewhat divergent from the rest of the OO community on the benefits (or otherwise) of polymorphism. Although it is useful to play devils advocate and question entrenched beliefs he fails to make a strong case for his point of view.

A.16 Dynamic modeling

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
The state models of subclasses should be consistent extensions of the state models of their superclasses. (Firesmith 1995)	PROD / PROC	AD-CO	Obj / Sub	meth

Table A-19 Dynamic modelling heuristics

The major point to be made about the heuristics for dynamic modelling in the literature is that there is an extreme dearth of them.

A.17 Miscellaneous

Heuristic	Prod/Proc/ Not/Man	Life-cycle Applicability	Assessment style	meth/lang dependent
Model the real world whenever convenient since this will provide a more natural platform for maintenance personnel. (Riel 1994)	PROD	AN-AD	Sub	
Beware of "proliferation of classes". Every added class increases system complexity and adds "uses". (Rising 1994)	PROD/PROC	AN-AD	Sub	
Distribute system intelligence as uniformly as possible. (Rising 1994)	PROD	AD-DD	Sub	
Decentralise control. Eliminate agents since many are irrelevant and don't do anything useful. (Rising 1994)	PROD	AD-DD	Obj / Sub	
Systems should be designed as networks of classes with no free-standing data or functions. (Korson and McGregor 1992)	PROD / PROC	AD-CO	Obj	
Distribute system intelligence across objects sharing lexical scope. (Riel 1994)	PROD	AN-AD	Sub	
Carefully examine system decomposition, paying closest attention to those parts where there are relatively small differences in the scales of time and/or size between a decomposed item and its components. (Berard 1993)	PROD	AD	Sub	
The ease of making a change to the design should bear a reasonable relation to the likelihood of the change being needed. (Parnas, Clements et al. 1983)	PROD/PROC	AD-CO	Sub	

Table A-20 Miscellaneous heuristics

Riel's statement 'model the real world whenever convenient' is rather vague. Just when is it convenient? Hawthorn suggests introducing new classes representing wider more general abstractions to aid maintainability (Haythorn 1994). Obviously there is a trade-off between these two points. Real world modelling aids maintainability by making the design easier to understand. Introducing new more general abstraction which do not exist in the problem domain makes the problem harder to understand but reduces the likely extent of changes that may need to be made.

Rising's recommendation to 'distribute system intelligence as uniformly as possible' is another way of saying that OO systems should be structured as a network of independent but co-operating objects.

A.18 Summarised heuristics

A.18.1 Introduction

The following set of tables represents a summary of the full heuristics and discussion given earlier. This sub-set of heuristics is aimed at architectural design in particular, therefore any of the previously given heuristics which do not apply to this stage are left out. The heuristics are similarly targeted at the products of design and not its process, notation or management. As only heuristics for products of the architectural design phase are now presented, the columns of the tables giving the lifecycle and heuristic type information are no longer necessary.

The original tables show much duplication of ideas between authors. Whilst in discussing the merits of a heuristic a knowledge how widely the heuristic is accepted is useful in trying to produce a single set of heuristics this duplication is superfluous. Duplicated heuristics have therefore also been removed. The intention is to leave only the most clear/concise description of the heuristic, or to replace the variations with a new but representative version.

A set of heuristics of this sort should be useful in assessing a design therefore a number of the original heuristics have also been removed because they were thought to vague to be usefully applied.

A.18.2 Inheritance

A.18.2.1 Depth of inheritance

The ideal depth of hierarchy seems to be based on a trade-off between reuse and understandability. Hierarchies shouldn't be forced to have a certain depth of hierarchy or to sit within a particular range if this would be inappropriate for the system in

question. However, a depth of hierarchy outside of the usual range would merit a re-evaluation of the design to verify that the hierarchy had been well structured. A suggestion for the usual range for depth of inheritance hierarchies is given in table below.

Number of classes	20	50	100	500
'Ideal' Depth of Hierarchy	1-4	3-6	5-9	7-10

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
1.1	Inheritance hierarchy depth should fall within the range shown above for the relevant size of system.	Obj / Sub	lack of reuse	
1.2	Inheritance hierarchies should be no deeper than the maximum depth shown above for the relevant size of system.	Obj / Sub	less understandable	

Table A-21 Depth of inheritance (summary)

A.18.2.2 When to use inheritance

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
2.1	Inheritance should only be used for subtyping (as opposed to implementation inheritance) (Bar-David 1992)	Obj / Sub		
2.2	The assertions of subclasses should be consistent extensions of the assertions of their parents. (Firesmith 1995)	Obj / Sub		
2.3	Replace implementation inheritance with aggregation except in the case of mixins. (Firesmith 1995)	Obj / Sub		
2.4	Don't confuse a class with the classes needed to implement it. Just because a class is implemented using other classes shouldn't mean it inherits from them. (Rumbaugh 1993)	Obj / Sub		
2.5	Use inheritance when the resulting subclasses will have different features and behaviours; otherwise use instances of power classes. (Firesmith 1995)	Obj / Sub		

2.6	Subclasses should be different from their superclasses. (Firesmith 1995)	Obj / Sub	10.3	
2.7	Do not model the dynamic semantic of a system as an inheritance hierarchy. (Riel 1994)	Obj / Sub		
2.8	Never create a new class if you can reuse an existing one. (Firesmith 1995)	Obj / Sub		
2.9	There shouldn't be any application specific methods in the specialisation interface. (Berard 1993)	Obj / Sub		
2.10	A subdimension should be a specialisation of the dimension it partitions. (McGregor and Korson 1993)	Obj / Sub		
2.11	All the methods in the specialisation interface should be appropriate for specialisations of the given object. (Berard 1993)	Sub		

Table A-22 When to use inheritance (summary)

As discussed earlier on of the consequences of applying the abstract superclass rule is that where a superclass would previously have been instantiated directly, an identical concrete copy of the now abstract superclass is instantiated instead. This clashes with rule 2.6 above.

A.18.2.3 What inheritance should model

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
3.1	An inheritance tree should be a generalisation-specialisation tree with more specialised capabilities near the leaves of the tree. (Love 1991)	Sub		
3.2	You will have little trouble with incorrect use of inheritance if an instance of the subclass is in every respect an instance of the superclass. (Rumbaugh 1993)	Sub	flexible implem-entation	
3.3	Specialisation inheritance is better than interface inheritance, which is better than implementation inheritance. (Firesmith 1995)	Obj / Sub		

Table A-23 What inheritance should model (summary)

A.18.2.4 Multiple inheritance

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
4.1	A subclass should never have more than two direct superclasses. (Firesmith 1995)	Obj	R.W.M.	lang
4.2	A mixin should be combined with a single generalisation to produce a specialisation with one or more focused capabilities. (Firesmith 1995)	Obj	Only one mixin?	lang
4.3	A class being defined via multiple inheritance should have at most one subclass in a given dimension. (McGregor and Korson 1993)	Obj / Sub		lang
4.4	Only use multiple inheritance for: • Specialisation inheritance • Adding capabilities using mixins. • For subclasses whose protocol is divided into separate roles. • For subclasses that are combining superclasses from two dimensions. • To separate interfaces from implementations. (Firesmith 1995)	Obj / Sub		lang
4.5	Watch out for the implementation of containment hierarchies as multiple inheritance. (Riel 1994)	Obj / Sub		lang
4.6	Try to reduce multiple inheritance to a single inheritance hierarchy by asking the question "are any of my base classes subclasses of one another?" (Riel 1994)	Sub		lang
4.7	When adding multiple inheritance to a OOD assume a mistake is being made and prove otherwise. (Riel 1994)	Sub		lang

Table A-24 Multiple inheritance heuristics (summary)

It is possible (if unusual) for a class to be a specialisation of more than two superclasses. Restricting the maximum number of superclasses, as suggested in 4.1 will prevent these classes being represented as they would be in real world modelling (RWM).

A.18.2.5 Miscellaneous inheritance

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
5.1	Use generics rather than inheritance where appropriate. (Firesmith 1995)	Obj / Sub		meth
5.2	Subclasses within a dimension should be defined so that any instance of the parent class clearly belongs to exactly one subclass. (McGregor and Korson 1993)	Obj / Sub		
5.3	Superclasses should standardise their subclasses. (Firesmith 1995)	Obj / Sub		
5.4	The extents of subclasses in a dimension should partition the extent of the superclass. (Firesmith 1995)	Obj / Sub		
5.5	The portion of a class' specification that is specialised to form the subclass within a dimension should be disjoint from the parts of the class' specification used to form other dimensions. (McGregor and Korson 1993)	Obj / Sub	possible lack of cohesion 12.6	
5.6 ??	Avoid mixing different levels of abstraction within the same model. (Rumbaugh 1993)	Sub		
5.7	Make the hierarchy's conceptual architecture explicit. (Dvorak 1994)	Sub		

Table A-25 Miscellaneous inheritance heuristics (summary)

In heuristic 5.5 above the recommended form for the class described is a set of disjoint sections. This would suggest that the class represents several disjoint concepts and may present several disjoint interfaces. Such a class would normally be considered to have poor cohesion.

A.18.3 Aggregation

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
6.1	Containing classes should know what they contain, but contained classes should have no knowledge of who contains them. (Riel 1994)	Obj		

6.2	Aggregation should not be used to represent non-compositional relationships, i.e., spatial/temporal inclusion, attribution, class membership, attachment, ownership. (Civello 1993)	Obj / Sub		
6.3	Designers should be clear about the precise relationship intended for each use of aggregation in a design. (i.e., visibility, separability, immutability, sharing, ownership) (Civello 1993)	Obj / Sub		
6.4	If a contained element has behaviour then at least some of that behaviour should be used by the containing object. (Riel 1994)	Obj / Sub		
6.5	Aggregation hierarchy depth should normally be approximately the same as inheritance hierarchy depth for systems of the same size.	Obj / Sub	reuse vs understandability	
6.6	Distribute system intelligence down the containment hierarchies. (Riel 1994)	Sub		

Table A-26 Aggregation heuristics (summary)

For aggregation depth as for inheritance depth their is a trade-off between reuse/reusability and understandability. The more factored the hierarchy the greater the reuse or potential reuse. The shallower the hierarchy the more easier it is to understand and therefore to test and maintain.

A.18.4 Associations

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
7.1	Both classes in an association have equal status. If one class is subordinate to the other then aggregation or attribution should be used. (Rumbaugh 1996)	Obj / Sub		
7.2	When considering an association or aggregation use the strongest form of relationship that is applicable.	Obj / Sub		

Table A-27 Association heuristics (summary)

A.18.5 Clusters

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
8.1	Classes should be grouped into defined collections of upto twenty classes that have restricted visibility from other classes in the system. (Love 1991)	Obj		lang meth
8.2	A "system of objects" should represent an object-oriented concept, as opposed to a functional concept. (Berard 1993)	Obj / Sub		
8.3	With a system of objects, all of the component objects should directly support the object-oriented concept that the system represents. (Berard 1993)	Obj / Sub		
8.4	If a system of objects presents multiple interfaces to the outside world, each of the interfaces should represent a complete and coherent object-oriented concept. (Berard 1993)	Obj / Sub	cohesion 12.6	lang meth

Table A-28 Cluster heuristics (summary)

In heuristic 8.4 above it is said that if an object has multiple interfaces each interface should represent a separate object-oriented concept. Heuristic 12.6 clearly states that any module should represent a single concept. A single object representing multiple concepts would normally be considered to have poor cohesion.

A.18.6 Classes

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
9.1	A class should have no more than six data members. (Riel 1994)	Obj		
9.2	A class which has no operations should not be a class (it is probably an attribute). (Riel 1994)	Obj		
9.3	If a class has only one or two operations then it should not be a class (it should probably be implemented as an operation on some other class). (Riel 1994)	Obj / Sub		

9.4	Avoid having too many services per class. Each class typically has no more than six or seven public services. (Coad 1991c)	Obj / Sub		
9.5	Given some desirable behaviour in a system, a decision must be made as to which class to place it in. The following criteria have been offered for consideration when making such a decision. • Reusability- would this behaviour be useful in more than one context. • Complexity- how difficult is it to implement the behaviour. • Applicability- how relevant is the behaviour to the type in which it might be placed. • Implementation knowledge- does the behaviour's implementation depend upon the inter details of a type. (Booch 1991)	Obj / Sub		
9.6	Avoid excessive attributes. On average only one or two attributes are needed per service. (Coad 1991c)	Obj / Sub		
9.7	Pre-ordained H/W components and their S/W interfaces are good candidates for system entities. (Coplien 1992)	Obj / Sub		
9.8	If you can't think of two good significantly different implementations, then a broader abstraction may be needed. (Coplien 1992)	Obj / Sub		
9.9	Semantic constraints on a class shouldn't be implemented in the class structure if this leads to a proliferation of classes. (Riel 1994)	Obj / Sub	R.W.M. 16.3	
9.10	Beware classes which have only one instantiation in a system. (Riel 1994)	Obj / Sub		
9.11	Beware of classes that are really operations. (Rising 1994)	Sub		
9.12	Avoid 'fuzzy' class definitions. A class should be cohesive with a single, well defined and clearly bounded purpose. (Coad 1991c)	Sub		
9.13	A class should have easily identifiable and nameable behaviours. (Coplien 1992)	Sub		
9.14	A class should not represent too large or too small an abstraction. (Coplien 1992)	Sub		

Table A-29 Class heuristics (summary)

Heuristic 9.9 is a pragmatic heuristic designed to prevent class proliferation. This is done however at the cost of real world modelling. In particular there is a trade off between this heuristic and 16.3, which states that if the value of a piece of data affects the behaviour of a class then inheritance should be used. The balance to be considered is how central the modelling of the semantic constraints is to the problem and how great a proliferation in classes they're modelling will cause.

A.18.7 Abstract classes

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
10.1	Use deferred classes to control future subclassing if appropriate. (Firesmith 1995)	Obj / Sub		
10.2	Abstract classes should capture common abstractions. (Firesmith 1995)	Sub		
10.3	Abstract superclass rule (ASR) states: all superclasses must be abstract. (Hursch 1994)	Sub	2.6	

Table A-30 Abstract class heuristics (summary)

The trade-off between 10.3 and 2.6 was discussed earlier.

A.18.8 Coupling

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
11.1	Minimise the tightness of those relationships that are necessary. (Myers 1978)	Obj / Sub		
11.2	Minimise the knowledge one class has of others, so when one class is changed, it is less likely to change the others. (Berard 1993)	Obj / Sub		
11.3	The larger the number of other distinct system components with which an object must deal directly, usually the poorer is the design of the overall system. (Berard 1993)	Obj / Sub		

11.4	Classes near the root of an inheritance tree should not depend upon classes further down the tree. (Love 1991)	Obj / Sub		
11.5	The coupling of objects should take place only on an application by application basis. (Berard 1993)	Obj / Sub		meth lang
11.6	Eliminate unnecessary relationships among modules. (Myers 1978)	Sub		
11.7	Avoid having a large, central object which controls most of the functionality. (Riel 1994)	Sub		

Table A-31 Coupling heuristics (summary)

A.18.9 Cohesion

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
12.1	An object containing only functions is not cohesive in an object-oriented sense. (Berard 1993)	Obj		
12.2	An object containing only data is not cohesive in an object-oriented sense. (Berard 1993)	Obj		
12.3	All encapsulated methods should be application-independent. (Berard 1993)	Obj / Sub		
12.4	Ensure all the member variables and methods of a class make sense as part of the class. (Berard 1993)	Sub		
12.5	Make sure that subsystems are comprised of parts that are all highly logically related. (Berard 1993)	Sub		
12.6	Any one module should represent a single, coherent concept, at a uniform level of abstraction. (Berard 1993)	Sub		
12.7	A software engineer should be able to understand the responsibility of a module without understanding the module's internal design. (Parnas, Clements et al. 1983)	Sub		
12.8	Each of the methods in the public interface for an object should perform a single coherent function. (Berard 1993)	Sub		

12.9	An object removed from the context of the immediate application, should, in isolation, still represent a coherent and complete object-oriented concept. (Berard 1993)	Sub		

Table A-32 Cohesion heuristics (summary)

A.18.10 Information hiding

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
13.1	Allow access to inherited state information only via methods. (Berard 1993)	Obj	execut. speed	
13.2	Variables should only be accessed or modified through the sending of messages to, or the invoking of, the accessing methods. (Wirfs-Brock and Wilkerson 1989)	Obj		
13.3	A change to a class should require the minimum number of design decisions in other classes to be reconsidered. (Coplien 1992)	Obj / Sub		
13.4	A type should be a logical black box specification of externally visible behaviour. (Rumbaugh 1993)	Obj / Sub		
13.5	It should be possible to change the implementation of one module without knowledge of the implementation of other modules and without affecting the behaviour of other modules. (Parnas, Clements et al. 1983)	Obj / Sub		

Table A-33 Information hiding heuristics (summary)

Introducing information hiding between a superclass and its subclasses as suggested in 13.1 has a slight execution speed trade-off. However, as previously discussed this will not normally cause significant problems.

A.18.11 Message passing

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent

14.1	Message passing may be used to : 1. request a corresponding service 2. provide notification of an event 3. provide data (actual parameters). (Firesmith 1993)	Obj / Sub		meth lang
14.2	Minimise object collaboration. In most cases objects should only collaborate with three to five other objects. (Coad 1991c)	Obj / Sub		
14.3	Minimise collaborators to 6 for a given system requirement. The overall number may be higher. (Rising 1994)	Obj / Sub		

Table A-34 Message passing heuristics (summary)

A.18.12 Services

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
15.1	Keep protocols as simple as possible, have no more than 6 data parameters per method. (Rising 1994)	Obj		
15.2	Any override must return the same value as the base class function would when applied to the base part of the derived class object. (Bar-David 1992)	Obj		lang
15.3	Usually if a service requires more than three parameters something is wrong. (Coad 1991c)	Obj / Sub		
15.4	An operation on a class should not have the potential for changing the semantics of behaviour of the class. (Coplien 1992)	Obj / Sub		
15.5	Services should do some work, not just pass the buck to another service. (Love 1991)	Obj / Sub		
15.6	Services of the same name in different classes should mean the same things. (Love 1991)	Sub		
15.7	Services of different names should mean different things. (Love 1991)	Sub		

Table A-35 Service heuristics (summary)

A.18.13 Attributes

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
16.1	Attributes should hold encapsulated, unshared values. (Rumbaugh 1996)	Obj		
16.2	Attributes should contain useful information about the objects of a class, but should not have associated behaviour. (Rising 1994)	Obj		
16.3	If the value of a piece of data affects the behaviour of a class then inheritance should be used.	Obj / Sub	9.9	
16.4	If a piece of data has associated behaviour use containment, otherwise, use an attribute.	Obj / Sub		

Table A-36 Attribute heuristics (summary)

The above trade-off was discussed with reference to heuristic 9.9.

A.18.14 Protocol

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
17.1	The protocol of a subclass should be a superset of the union of the protocols of its parents. (Firesmith 1995)	Obj / Sub		
17.2	Implement a minimal public interface which all classes understand. (Riel 1994)	Obj / Sub		
17.3	Keep protocol as simple as possible. (Riel 1994)	Sub		
17.4	The public interface for an item should be consistent with the overall concept embodied by the item. (Berard 1993)	Sub		

Table A-37 Protocol heuristics (summary)

A.18.15 Polymorphism

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent

18.1	To localise the effect of changes and hence reduce maintenance effort the use of polymorphism is essential. (Haythorn 1994)	Obj / Sub		
18.2	Great classes are essentially interface specifications embodied as abstract classes. We don't create the abstract class because it is evident in the domain - we create it to control the impact of future changes in the domain. (Haythorn 1994)	Sub	20.3	

Table A-38 Polymorphism heuristics (summary)

Heuristic 18.2 suggests selecting classes to enable control future changes to the domain, rather than to model its current state as suggested in 20.3.

A.18.16 Dynamic modelling

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
19.1	The state models of subclasses should be consistent extensions of the state models of their superclasses. (Firesmith 1995)	Obj / Sub		meth

Table A-39 Dynamic modelling heuristics (summary)

A.18.17 Miscellaneous

Heuristic Number	Heuristic	Assessment Style	Trade-offs	meth/lang dependent
20.1	Systems should be designed as networks of classes with no free-standing data or functions. (Korson and McGregor 1992)	Obj		
20.2	Decentralise control. Eliminate agents since many are irrelevant and don't do anything useful. (Rising 1994)	Obj / Sub		
20.3	Model the real world whenever convenient since this will provide a more natural platform for maintenance personnel. (Riel 1994)	Sub	18.2	
20.4	Beware of "proliferation of classes". Every added class increases system complexity and adds "uses". (Rising 1994)	Sub		

20.5	Distribute system intelligence as uniformly as possible. (Rising 1994)	Sub		
20.6	Distribute system intelligence across objects sharing lexical scope. (Riel 1994)	Sub		
20.7	Carefully examine system decomposition, paying closest attention to those parts where there are relatively small differences in the scales of time and/or size between a decomposed item and its components. (Berard 1993)	Sub		
20.8	The ease of making a change to the design should bear a reasonable relation to the likelihood of the change being needed. (Parnas, Clements et al. 1983)	Sub		

Table A-40 Miscellaneous heuristics (summary)

Blank

In

Original

Appendix B Measurement theory

Measurement theory is a branch of mathematics the fundamental concepts of which are based on the model theory of logic. It outlines how valid measures can be defined and underpins the types of data analysis that are meaningful for particular measures.

B.1 Representational theory

The basic precept of measurement theory is that the numbers or symbols assigned to entities must represent the perceived relations between the attributes of those entities. The three main components necessary in defining a measure are an empirical relational system, a representational theorem, and a uniqueness condition.

B.1.1 Relational systems

The standard definition of a relational system **A** is an ordered pair (A,R). Where A is a set of objects and R is a set of n-ary relations defined on those objects. There is a variation on this definition which defines **A** as an ordered triple (A,R,o) where o is a set of binary operations defined on A. These two definitions are equivalent because in the first definition R subsumes o. In the second they are identified separately. The following discussion uses the first definition.

Measures require the definition of two relational systems, an empirical relational system and a formal (usually numerical) relational system.

B.1.2 Empirical relational system

An empirical relational system, $\boldsymbol{Q}$, may be defined as follows:

$\boldsymbol{Q} = (Q,R)$ where,

Q is a set of real world entities (objects/events/processes) which we wish to measure.

$R = \{R_1, R_i, R_n\}$ is a set of relations defined on Q. This set of relations is our perception of how the entities behave, and how they may be manipulated and compared.

B.1.3 Formal relational system

A formal relational system, *N*, may be defined as follows:

$N = (N,P)$ where,

N is a set of formal entities (symbols/numbers)

$P = \{P_1, P_i, P_n\}$ is a set of relations defined on N.

B.1.4 Representational Theorem

Measurement, *M*, is a systematic process of mapping an entity's attribute onto a formal representation $M{:}Q \rightarrow N$. This mapping must be done in such a way that the relations between the empirical world entities must hold between their formal world equivalents. This is the representation theorem and may be more formally described as follows:

$$R_i(q,r,\ldots) \Leftrightarrow P_i[M(q), M(r),\ldots] \quad \forall i = 1,n$$

where,

P_i is a relation on N corresponding to R_i, and $q,r,\ldots, \in Q$

B.1.5 Uniqueness Condition

The conditions under which the representation theorem holds are termed the representation conditions. In general if the mapping, *M*, satisfies these conditions they will be other mappings, *M'*, which will also satisfy these conditions. These other mappings may be related to the original by a transformation, *f*, i.e. $M' = f(M)$. As these other mapping also hold to the representation conditions it follows that:

$$R_i(q,r,\ldots) \Leftrightarrow P_i[f(M(q)), f(M(r)),\ldots]$$

Transformations for which the above holds are termed admissible transformations. The set of all such admissible transformations are known as the uniqueness condition.

B.1.6 Scales

A scale is the term given to the triple of empirical relational system, formal relational system and a mapping. Following the above definitions, a scale, *S*, is defined as:

$S = (Q, N, M)$. Although in theory each triple has its own scale with it's own uniqueness condition, in practice scale types are generally considered rather than individual scales. A scale type encompasses a range of different scales that share a set of admissible transformations.

B.2 Scale types

The most commonly used scale types and their 'admissible' transformation are described in Table B-1.

Scale Type	Admissible Transformations	Examples
Nominal	$M' = f(M)$ where f is a 1-1 mapping	classification (fault types, development languages etc.)
Ordinal	$M' = f(M)$ where f is monotonically increasing	preferences (more user friendly, more maintainable)
Interval	$M' = \alpha M + \beta$ ($\alpha > 0$)	temperature (centigrade and Fahrenheit), time (calendar)
Ratio	$M' = \alpha M$ ($\alpha > 0$)	length (meters), mass (Kg), energy (J)
Absolute	$M' = M$	counting (number of children, depth of inheritance)

Table B-1 Scale types and admissible tranformations

Table B-2 lists the appropriate statistics and tests for the standard scale types.

Scale Type	Defining Relations	Appropriate Statistics	Appropriate tests
Nominal	Equivalence.	Mode, frequency, and contingency coefficients	Non-parametric statistical tests
Ordinal	Equivalence, greater than.	Median, percentile, Kendall τ, Spearman r_s, *and Kendall W*	
Interval	Equivalence, greater than and known ration of any intervals.	Mean, standard deviation, pearson product-moment correlation, and multiple product-moment correlation	Non-parametric, and parametric statistical tests
Ratio	Equivalence, greater than and known ration of any intervals, and known ratio of any two scales	Geometric mean and coeffiecient of variation	

Table B-2 Scale type and appropriate statistics

Blank

In

Original

Appendix C Meta-modelling object-oriented designs

The purpose of the data model is to represent the information contained within a CASE tool output, or a programming language file, in a form suitable for storage on a relational database. This database must be capable of supporting queries designed to calculate OO design metrics.

C.1 Inputs to the model

A fundamental consideration in the design of the model is that it must be capable of storing the design information about an OO system. One problem in designing such a model is that, although the general concepts behind OO design are fairly well understood, the detailed way in which the OO principles are implemented varies between particular programming languages and particular design methods. Building a tool that takes input from a variety of sources requires a common model that captures as much of the conceptual detail as possible across the more common languages and methods. In doing this, there is an inevitable loss of precision in the description of some concepts from the original notations.

A desire to produce a single standard notation to describe the OO designs recently emerged in the OO community in general. This has resulted in two major unification efforts between the existing methodologists. These are the unified modelling language (UML) and the OPEN method. UML consists of a notation and a description of the semantics of the notation. The UML semantics has been accepted as the basis for a common OO meta-model by the Object Management Group (OMG) and as such is likely to become the de facto standard. For this reason the semantics was used as the principle input for the design of the data model.

As the purpose of the model is to allow metrics to be collected, it must include any data that is required to calculate known metrics. A second source of input for the data model must therefore be the knowledge of what information is required to assess

existing metrics.

Where information required for a particular metric is not present in the UML meta-model (and the information is available at design time) this information was used to extend the data model beyond UML.

An additional source of input for the derivation of the model are additional requirements imposed on the model because of the desired functionality of the tool.

C.2 The UML meta-model

The UML meta-model is structured as three top-level packages Foundation, Behavioural Elements and Model Management.

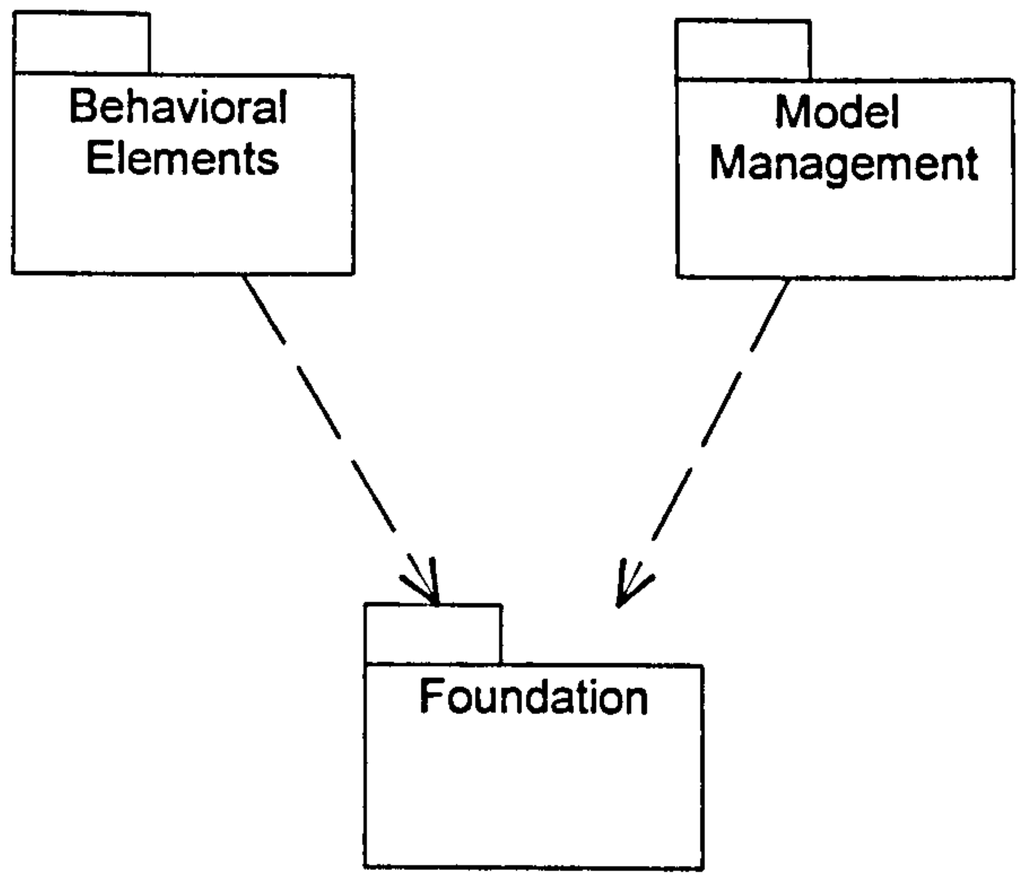


Figure C-1 UML top-level packages

The Foundation package is further decomposed into four sub-packages Auxiliary elements, Core, Extension Mechanisms and Data Types.

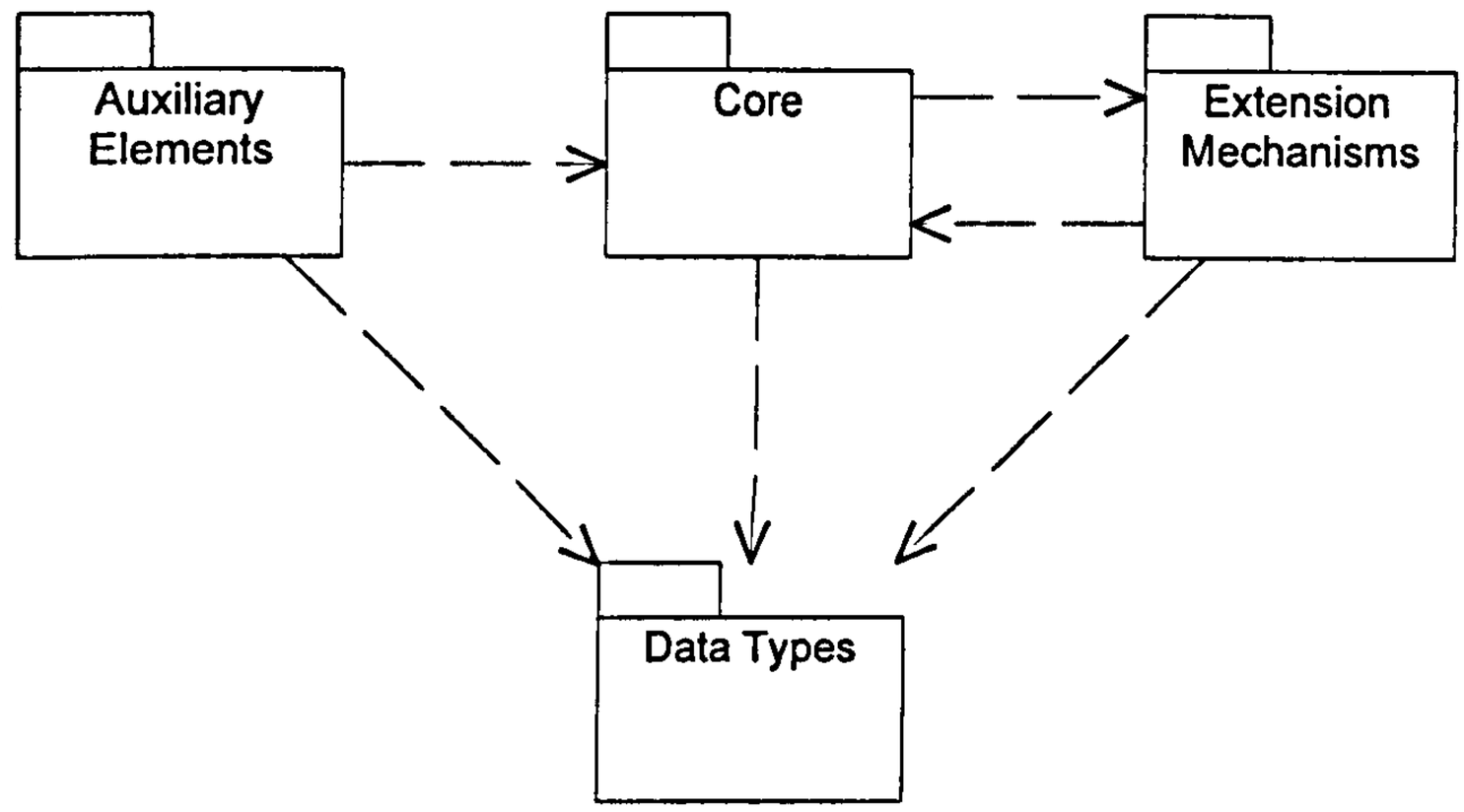


Figure C-2 UML foundations package

The Behavioural package is also decomposed into four sub-packages Collaborations, Use Cases, State Machines and Common Behaviour.

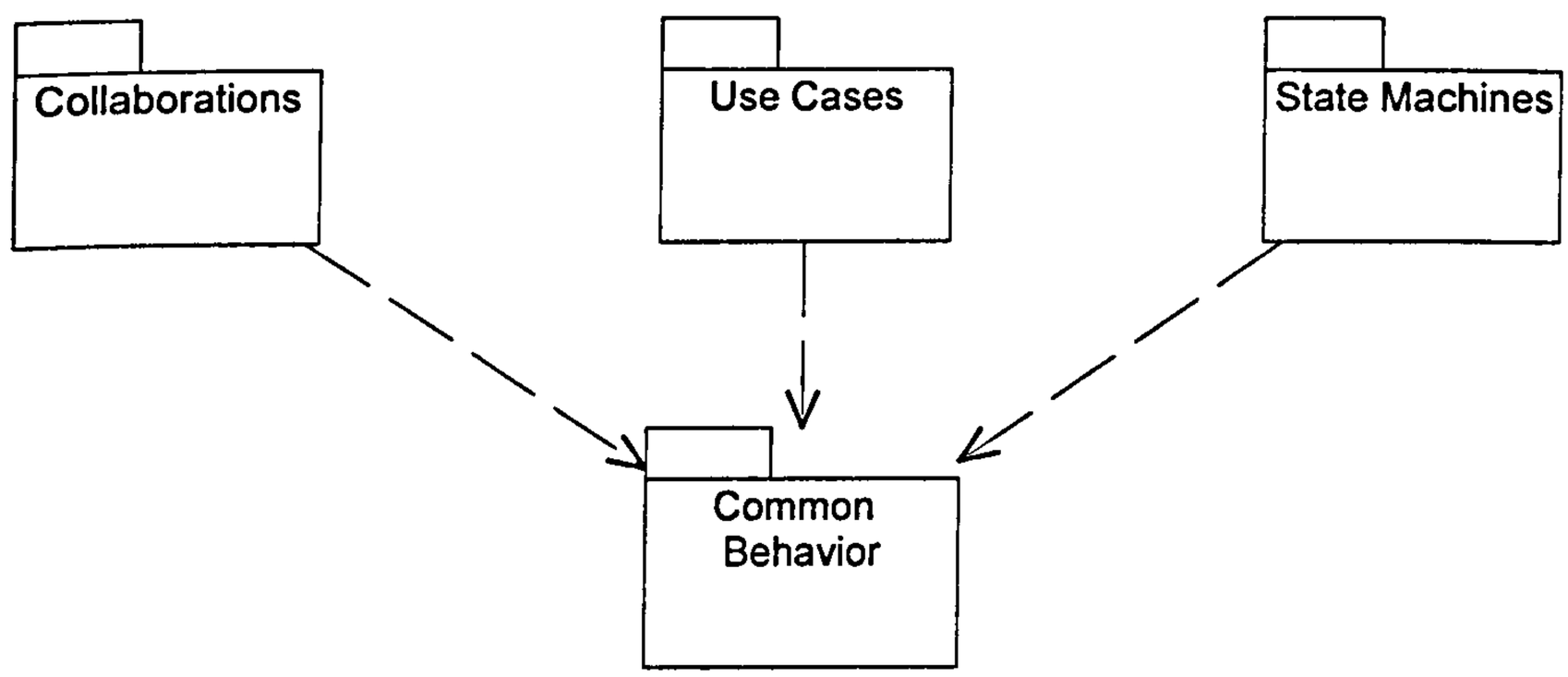


Figure C-3 UML behavioural package

Each of the packages used to describe the UML meta-model contain only those attributes and relationships which are relevant to that particular package. This means that to get a complete picture of any class a union of all attributes and relationships must be made across all of the packages in which that class appears. A piecemeal approach to producing a relational model from the UML meta-model is therefore fraught with problems of having to continually re-work previous packages as new relationships are introduced in later packages which invalidate the mappings already made.

The approach taken to producing a relational model from the UML meta-model is therefore to first produce a single meta-model diagram showing the complete classes from all packages. As mentioned earlier there are additional considerations in producing the relational model. What information about a design needs to be stored in order for known metrics to be calculated? How will CASE tool and language mappings be achieved? How the resulting database will be constructed and maintained? How the database will be used within the tool? These considerations will be used to modify the meta-model to better suit it's purpose before the relational model if finalised.

C.3 Metrics considerations

In this section the elements of the UML meta-model that are necessary for the calculation of existing measures is considered. One obvious consideration is whether the dynamic elements of the meta-model are necessary. Design and coding measures are usually based on the static aspects of a system. There is little reference in the literature to measuring the dynamic aspects of system. Collecting truly dynamic measures would require instrumenting and running the code (or executable design). Dynamic measures would require a very different type of tool from the static analysers normally used for measurement collection. It was therefore decided to limit the tool to the static aspects of the UML meta-model.

Some measures described as being design metrics require information not normally contained in design diagrams. For instance, to calculate LCOM information on which class methods use which class attributes is required. This information is not generally available in UML design diagrams. However, the meta-model does have a stereotyped dependency relationship called 'usage' that could be used to represent these method-to-attribute relationships. This means that where these relationships exist in code they could be modelled, but it unlikely that they would be represented in design diagrams.

A more general method for considering which elements of the meta-model would be

useful in supporting measurement is to look at a collection of measures and consider which meta-model elements would be necessary for their collection. The table below lists some commonly used design/code measures and gives the meta-model elements that they would be necessary for their calculation.

Originator	Metric	UML Meta-class	Attribute or Relationship
Chidamber & Kemerer	WMC	Class	features
	DIT	GeneralizableElement	isRoot, supertype
	NOC	GeneralizableElement	isLeaf, subtype
	CBO	«call» Dependency «usage» Dependency	client, supplier client, supplier, description
	RFC	call «Dependency»	client, supplier
	LCOM	«usage» Dependency ModelElement	client, supplier type
Lorenz & Kidd	Number of scenario scripts		
	Number of key classes		
	Number of support classes		
	Average number of support classes per key class		
	Number of subsystems	Classifier	type
	Average per-days per class		
	Average number of classes per developer		
	Number of major iterations		
	Number of contract completed		
	Number of message sends		
	Number of statements		
	Lines of code		
	Average method size		
	Method complexity		
	Strings of message sends		
	Number of message sends	«call» Dependency	client, supplier
	Number of public instance methods	Classifier Feature	features, visibility, type
	Number of instance methods	Classifier Feature	features scopeKind, type

	Average number of instance methods per class	Classifier Feature Package	features visibility, type ElementReference
	Number of instance variables	Classifier Feature	features scopeKind, type
	Number of class methods	Classifier Feature	features scopeKind, type
	Number of class variables	Classifier Feature	features scopeKind, type
	Class hierarchy nesting level	GeneralizableElement	isRoot, supertype
	Number of abstract classes	GeneralizableElement	isAbstract, type
	Use of multiple inheritance	GeneralizableElement	supertype
	Number of overridden methods	Classifier Feature BehavioralFeature GeneralizableElement	features type, name parameters supertype
	Number of methods added by a subclass	Classifier Feature BehavioralFeature GeneralizableElement	features type, name parameters supertype
	Specialization index	Classifier Feature BehavioralFeature GeneralizableElement	features type, name parameters isRoot, supertype
	Class cohesion		
	Global usage		
	Average number of parameters per method	BehavioralFeature	parameters
	Use of friend functions	«friend» Dependency ModelElement	client, supplier type
	Percentage of function-oriented code		
	Average number of comment lines per method		
	Average number of commented methods		
	Number of problem reports per class or contract		

Table C-1 Meta-model element usage by measures

Those elements of meta-model used in the collection of the listed measures will clearly be necessary parts of the tool's meta-model. Two other points are also clear from the table. Firstly, some of the measures require information for their calculation that are not available from the UML meta-model. Should the tool's meta-model be

augmented to include this information? Alternatively, should these measures be reclassified as code-only measures and so not collectable by a design measurement tool? A second point is that some of the information included in the UML meta-model is not used in any of the measures listed. If elements of the UML meta-model are not used by measures, they might be removed to simplify the tool's meta-model. However, the list of measures is clearly not exhaustive, and even if it was, thought might be given as to whether future measures would use additional UML meta-model elements. Care should therefore be exercised in removing meta-model elements.

A further consideration is raised by the 'percentage of function-oriented code' metric. This measure assesses the amount of not-OO code. To calculate the measure the meta-model must be capable of representing systems which are only partially object-oriented (as is possible in C++). Should the tools meta-model be capable of storing information about the function-oriented sections of C++ programs, or should such measures not be considered as OO design measures and hence not included in an OO design measurement tool?

Another question is the extent to which language specific features should be included in the meta-model. For instance, should C++ friend classes/functions or Java inner-classes be included? There is a clear trade-off between the completeness of the meta-model and it's complexity.

C.4 Database considerations

1. Operation names require function names to be unique (for the primary key)- use function mangling
2. Consider how to handle long identifier names correctly? - limit to a sensible size, then truncate (or truncate and number as with shortened file names)
3. How can you handle the relational definition of metrics that employ recursion? - use 'connect by' in oracle (Access/other?)

C.5 Analysis tool considerations

1. Maintain a version attribute for designs in progress to allow multiple versions to

be stored and 'change over time' metrics to be calculated.

2. Add source file date-stamp to type-info in the database to facilitate minimum re-*building?*
3. Add a 'source' table to cover the last 2 points. This table will include:
   - name
   - namespace
   - filename
   - version number
   - date stamp
4. Add filename to the primary key of any ModelElement subtypes and update any foreign keys referring these appropriately.

C.6 Modelling considerations

C.6.1 Paring the UML meta-model

The UML meta-model contains information not only about the design model but also about how this model should be displayed. This information is useful to those using the meta-model description to help them in developing UML CASE tools but it is only the model information itself and not the view information which is of interest for the data model. For this reason the 'view elements' and their associated constructs will not be included when converting the UML meta-model into an database schema.

C.6.2 Converting UML to an RDBMS schema

The UML meta-model on which the data model is based is itself described in UML notation. As an OO design notation UML contains such constructs as inheritance and association, as well as concepts such as polymorphism, which are cannot be directly modelled in an RDBMS schema. The UML meta-model being described using the UML notation means that the conversion of these OO constructs needs to be handled at two levels. The first of these being the at the level of the model itself and the second at the level of the data which the model will contain.

Rules have already been suggested for converting OO models into RDBMS schema e.g., (Blaha, Premerlani et al. 1994). The following points should be considered when mapping from OO models to relational schema.

C.6.2.1 Classes

In general the classes in a model map to the entities in the relational schema. However, there are cases when more than one class may be combined into a single entity, or a single class split so that it is represented by two or more entities.

Considerations:

1. Where one class has a fixed cardinality association (e.g. one-to-one) with another class (especially where it is a containment association) it may be possible to combine the two classes into one entity.
2. Entities must contain a fixed number of simple attributes. Where a class contains complex attributes , e.g., instances of other classes as attributes, it may have to be split into two or more entities. Similarly, if a class contains a variable number of attributes, e.g., by containing a Vector, then it may also have to be split.

C.6.2.2 Template classes

1. Generally each instance of a template class must be treated as a separate entity. However, there are circumstances in which different template instances may be represented by the same entity. Some of these circumstances are:
2. If template parameters only affect the operations of the template instances.
3. If a class is parameterized using a value rather than a type and this doesn't affect the number of attributes of the class.
4. If a class is parameterized using a value rather than a type which affects the number of attributes of the class, e.g., by setting an array size, but this variable sizing is dealt with by using a separate entity for the variable attribute (as discussed above).

C.6.2.3 Attributes

1. Attributes in classes are generally mapped as equivalent attributes in their classes

associated entity. As mentioned above for this direct mapping to take place the attribute must be a single simple value. Composite attributes, or multiple valued attributes (arrays), may be modelled as either separate entities or several single attributes in their class's associated entity.

C.6.2.4 Operations

1. Operations are not generally mapped from object models to entity models.
2. Possibility of mapping operations that represent derived attributes?

C.6.2.5 Associations

1. Simple associations (including aggregation) may be implemented in either of two ways.
2. Creating an explicit table for the association. This table will include foreign keys for each of the associated classes and columns for any link attributes.
3. Including the primary key of one of the associated classes as a foreign key in the other. Any link attributes must also be included along with the foreign key.

Considerations:

1. Creating explicit tables for associations makes the schema more extensible but will generally reduce performance
2. When considering which form of mapping to use the multiplicity of the association must be taken into consideration.
3. An association cannot be made part of an entity which connects to many other objects via this association.
4. A many-to-many association must be implemented as a separate entity.
5. In a class whose association-role has a lower cardinality limit of one any attributes representing the association may not be null.
6. Where a separate table is used to represent an association a lower limit of one may not be enforced.
7. Associations between more than two classes are best modelled with a separate association entity.

C.6.2.6 Composition

1. Composition is a stronger form of association and as such obeys the mapping rules for associations.

C.6.2.7 Link classes

1. Link classes may be considered as a collection of link attributes and treated in the same way. Alternatively they may be mapped to a separate entity with relationships to each of the other classes involved in the association. Which is chosen will depend on the number of classes involved in the association and their various multiplicities.

C.6.2.8 Inheritance

1. Inheritance represents the largest problem in mapping object models into relational schema. Mapping of inheritance can be done in a number of ways. The primary consideration on the choice of mapping for inheritance is the manner in which the various superclasses and subclasses will be used in querying the resulting database. This basically means the extent to which polymorphism is used.
2. Use only a single superclass table. Attributes from each subclass are pushed up into the superclass. An additional attribute is also added to indicate the actual class to which a particular entry belongs. All attributes not applicable to a particular entry are set to null. This mapping violates second normal form because some attributes are dependent only on the type discriminator and not the full primary key. Also, it is not possible for most RDBMSs to enforce permissible permutations of null and non-null values. This is probably the easiest mapping to use with polymorphism. It will also have the best performance for use with polymorphism because only a single table need be searched and no joins are required. However, where deep hierarchies are used the necessary redundancy may be considered excessive. There may also be problems if partial hierarchy queries are required for hierarchies of depth greater than one. This is due to the subtypes only 'knowing' the root class of the hierarchy and not the intermediate levels of it's definition.

3. Use only subclass tables. Superclass attributes are pushed down into the subclasses. In this mapping it is not possible for most RDBMSs to constrain values across tables. For situations where polymorphism is not needed, i.e., where searches only occur within a specific predefined class, this mapping is probably the simplest and will give the best performance. This is because only a single table need be searched, no joins are required and no data redundancy is needed.
4. Use one entity for the superclass and an entity for each subclass. All entities must share a common primary key. Referential integrity can be used to ensure that each subclass record has a corresponding superclass record. This provides a compromise solution between 1 and 2. Superclass and subclass tables may be joined to allow polymorphic searches whilst each individual subclass also has it's own table for searches where the subclass is predefined.
5. Use one entity for the superclass, an entity for each subclass and a generalisation table. The generalisation table connects the primary key of the superclass to the primary keys of each of its subclasses. Referential integrity can be used to ensure that each subclass record has a corresponding superclass record. This type of mapping allows polymorphic queries without data redundancy where a common primary key between subtype and supertype does not exist.

C.6.2.9 Standard Elements

1. Consider how to deal with standard elements
2. How many of the standard element should be included in the model? All, only a selection , or none. If only a selection what would be the selection criteria?
3. How should included standard elements be modelled?
4. As subtypes of the Elements to which they apply?
5. As attributes of the Elements to which they apply?
6. Another form of mapping?
7. Might the mapping style be different for each of the three types of standard elements? (Stereotype, TaggedValue and Constraint)

C.7 Mapping the composite meta-model onto the relational model

C.7.1 Simplifying the UML structural packages

1. Meta-classes removed as they are solely concerned with the display of models and not with their content: Geometry, GraphicMarker, ViewElement, Presentation
2. Meta-classes removed as they have no semantic impact: Mapping
3. Remove EnumerationLiteral as it is closer to a code detail that a piece of design.
4. Collapse Primitive, Enumeration and Structure into stereotypes of DataType as they now contain no additional attributes or relationships.
5. Although shown on the diagram as a subclass of ModelElement it is described as being a subclass of ViewElement. As a ViewElement a Comment is not part of the model itself. Even if considered as a ModelElement comments may not be interpreted by a measurement system and so have no semantic impact. Remove meta-class Comment.
6. Collapse Refinement, Trace and Usage into stereotypes of Depencency
7. Collapse Model into a stereotype of Package
8. Collapse Interface into a stereotype of Classifier
9. Collapse ProcedureExpression, TimeExpression, ObjectSetExpression and BooleanExpression into stereotypes of Expression.
10. Remove Element as it has no attributes or associations and there is only one remaining subclass, ModelElement, as we are no longer considering ViewElement.
11. Remove the Extension mechanisms. Extensions to the meta-model would require extensions to the database, the mappings and possibly existing metric scripts. The standard elements defined in UML will be incorporated into the model but additional extensions will not be allowed.
12. Add new StereoTypes : Primitive, Enumeration, Structure, Refinement, Trace Usage, Model, Interface, ProcedureExpression, TimeExpression, ObjectSetExpression, BooleanExpression
13. The aggregation relationship between Multiplicity and MultiplicityRange allows any set of non-negative integers to be constructed for a Multiplicity value. In practice, however, is rare to have other than a single contiguous range. Most

CASE tools will also only allow a single lower to upper range. For this reason the Multiplicity and MultiplicityRange meta-classes will be merged into a single Multiplicity meta-class with integer attributes of upper and lower.

14. Name with a single body attribute can be similarly substituted in the classes which contain it (ModelElement, ElementReference, Generalization and Expression). Addition constraints may be necessary to ensure that these new String attributes refer to names in the system.

C.7.2 Mapping the simplified UML structural packages into ERDs

C.7.2.1 Mapping associations and aggregations

1. As Multiplicity now has a fixed number of attributes it can itself be removed and these attributes (upper and lower) used to substitute for it in the meta-classes which contain multiplicities (AssociationEnd and StructuralFeature).
2. The composition of an dependency owningDependency to a subDependency may be modelled by an owningDependency attribute in the Dependency class. Since the cardinality is 0..1 this attribute may optionally be NULL.
3. The specification-realisation association between Operation and Method can be represented as an operation attribute in Method.
4. The parameter association between Parameter and BehavioralFeature can be represented by a behavioralFeature attribute in the Parameter class. As this association is ordered it will also require a parameter number attribute.
5. The type association between Parameter and Classifier can be represented by a type attribute in the Parameter class.
6. The featue-owner association between Feature and Classifier can be replaced by an owner attribute in the Feature class. Although this realtionship is given the {ordered} constraint there is no apparent reason for this (and no reason is given in the UML documentation). There seems no reason, therefore, for any additional attributes to record the ordering of features.
7. The type-feature association between Classifier and StructuralFeature can be replaced by a type attribute in the StructuralFeature class. Again there is no

apparent reason to retain any ordering information.

8. The ownedElement-namespace association between ModelElement and Namespace can be replaced by adding attributes for namespace and visibility (the ElementOwnership link-class's attribute) into the ModelElement class.
9. The gerneralization-subtype and the specialization-supertype association between Generalization and GeneralizableElement can be mapped as attributes in the Generalization class.
10. The aggregation link between Association and AssociationEnd can be replaced by adding association and endNumber attributes to AssociationEnd.
11. The template-templateParameter aggregation between ModelElement and itself must be replace by a TemplateParameter class because a relationship between an entity and many instances of itself cannot be represented by merely adding attributes. The TemplateParameter class has attributes: templateElement, which represents the ModelElement which is the template itself; parameterType which represents the ModelElement which specifies the type for the template argument of Binding; and parameterNumber, which holds the position in the parameter list of this particular parameter.
12. Map association as a class called TemplateArgument with attributes binding, type and argumentNumber.
13. The specification-realization association between Classifier and itself is replaced with an Implements class (must be a class because of many-to-many relationship).
14. The aggregation of ModelElement into Component appears to a restatement of the specification-realization relationship between Clasifier and itself. It will therefore be removed.

C.7.2.2 Collapsing the inheritance hierarchy

1. ModelElement is used polymorphically in Dependency, TemplateParameter and ElementReference. This means that mapping 2 cannot be used. The depth of ModelElements hierarchy (5) and the large number of subclasses and hence large number of attributes make mapping 1 impractical. If we make the assertion that a ModelElement's name should be unique with it's namespace then the name-namespace pair can be used as the primary key for all ModelElements (what ever

their subtype). This would allow mapping type 3 to be employed.

2. Make Binding a stereotype of Dependency rather than an explicit subclass.
3. Make Component, Node and DataType stereotypes of Classifier
4. Add Stereotypes: binding, component, node and dataType.
5. Only 2 attributes exist in the subclasses under GeneralizableElement. Mapping 1 can be used as polymorphism is required but the small number of subclass attributes would lead to little data redundancy.
6. Add class, associationClass, association, package and subsystem as stereotypes of GeneralizableElement. It is not necessary to add a stereotype for Classifier as this class is abstract and will never have any instances.
7. As StructuralFeature is abstract and has only one subclass, mapping 2 will be used to collapse the StructuralFeature-Attribute generalisation.
8. BehavioralFeature is used polymorphic by Parameter. Here mapping 1 can be used to collapse Operation and Method into BehavioralFeature.
9. Use mapping type 1 to collapse Namespace and GeneralizableElement.
10. As Feature is an abstract class and there seems no need for polymorphism mapping type 2 with be used.

Appendix D Design description language (DDL)

The following is a BNF description of the intermediate representation used to describe OO designs for input to the 'DesignDatabase' datastore.

Literal strings are in **bold**

Literal characters are in single quotes, e.g., 'b'

DefinitionFile ::= ElementDefinitions

ElementDefinitions ::=
/* EMPTY */
| ElementDefinitions ModelElementDefinition
| ElementDefinitions TemplateDefinition

ModelElementDefinition ::=
GeneralizableElementDefinition
| MethodDefinition
| AttributeDefinition
| ElementReferenceDefinition
| ImplementsDefinition
| DeploymentDefinition
| GeneralizationDefinition
| DependencyDefinition
| AssociationEndDefinition
| ParameterDefinition

GeneralizableElementDefinition ::=
<GeneralizableElement>
<Name> Name
<Namespace> Namespace
<Type> Type
<Stereotype> Stereotype
<Visibility> VisibilityKind
<IsAbstract> BooleanLiteral
<IsActive> BooleanLiteral
<IsInstantiable> BooleanLiteral
<IsLeaf> BooleanLiteral
<IsPersistent> BooleanLiteral
<IsRoot> BooleanLiteral

</GeneralizableElement>

MethodDefinition ::=
<Method>
<Name> Name
<Namespace> Namespace
<Type> Type
<Stereotype> Stereotype
<Concurrency> CallConcurrencyKind
<OwnerScope> ScopeKind
<Visibility> VisibilityKind
<IsPolymorphic> BooleanLiteral
<IsQuery> BooleanLiteral
</Method>

ParameterDefinition ::=
<Parameter>
<Name> Name
<Namespace> Namespace
<Stereotype> Stereotype
<DefaultValue> DefaultValue
<ParameterDirectionKind> ParameterDirectionKind
<ParameterNumber> ParameterNumber
<ElementType>
  <Name> Name
  <Namespace> Namespace
</Parameter>

AttributeDefinition ::=
<Attribute>
<Name> Name
<Namespace> Namespace
<Stereotype> Stereotype
<Changeable> ChangeableKind
<InitialValue> InitialValue
<Multiplicity> Multiplicity
<OwnerScope> ScopeKind
<Visibility> VisibilityKind
<ElementType>
  <Name> Name
  <Namespace> Namespace
</Attribute>

ElementReferenceDefinition ::=
<ElementReference>
<Alias> Alias
<Stereotype> Stereotype
<Visibility> VisibilityKind

<Element>
<Name> Name
<Namespace> Namespace
<Type> Type
<Package>
<Name> Name
<Namespace> Namespace
</ElementReference>

ImplementsDefinition ::=
<Implements>
<Stereotype> Stereotype
<Realization>
<Name> Name
<Namespace> Namespace
<Type> Type
<Specification>
<Name> Name
<Namespace> Namespace
<Type> Type
</Implements>

DeploymentDefinition ::=
<Deployment>
<Stereotype> Stereotype
<Component>
<Name> Name
<Namespace> Namespace
<Node>
<Name> Name
<Namespace> Namespace
</Deployment>

GeneralizationDefinition ::=
<Generalization>
<Name> Name
<Namespace> Namespace
<Discriminator> Discriminator
<Stereotype> Stereotype
<Depth> Integer
<Subtype>
<Name> Name
<Namespace> Namespace
<Type> Type
<Supertype>
<Name> Name
<Namespace> Namespace

<Type> Type
</Generalization>

DependencyDefinition ::=
<Dependency>
<Name> Name
<Namespace> Namespace
<Type> Type
<Stereotype> Stereotype
<Client>
<Name> Name
<Namespace> Namespace
<Type> Type
<Owner>
<Name> OwnerName
<Namespace> OwnerNamespace
<Supplier>
<Name> Name
<Namespace> Namespace
<Type> Type
<Dependency>

AssociationEndDefinition ::=
<AssociationEnd>
<Name> Name
<Namespace> Namespace
<Stereotype> Stereotype
<AggregationKind> AggregationKind
<ChangeableKind> ChangeableKind
<Multiplicity> Multiplicity
<TargetScope> ScopeKind
<IsNavigable> BooleanLiteral
<IsOrdered> BooleanLiteral
<Association>
<Name> Name
<Namespace> Namespace
<ElementType>
<Name> Name
<Namespace> Namespace
</AssociationEnd>

TemplateDefinition ::=
TemplateParameterDefinition
| TemplateArgumentDefinition

TemplateParameterDefinition ::=
<TemplateParameter>
<Type> Type

<Stereotype> Stereotype
<ParameterNumber> ParameterNumber
<ElementType>
  <Name> Name
  <Namespace> Namespace
<Template>
  <Name> Name
  <Namespace> Namespace
</TemplateParameter>

TemplateArgumentDefinition ::=
<TemplateArgument>
<Stereotype> Stereotype
<ArgumentNumber> ArgumentNumber
<Binding>
  <Name> Name
  <Namespace> Namespace
<ElementType>
  <Name> Name
  <Namespace> Namespace
</TemplateArgument>

ParameterDirectionKind ::=
#in
| **#out**
| **#inout**
| **#return**
| **#unknown**

ScopeKind ::=
#classifier
| **#instance**
| **#unknown**
| **#notApplicable**

VisibilityKind ::=
#public
| **#protected**
| **#private**
| **#implementation**
| **#unknown**
| **#notApplicable**

AggregationKind ::=
#none
| **#shared**
| **#composite**

| **#unknown**
| **#notApplicable**

ChangeableKind ::=
#none
| **#frozen**
| **#addOnly**
| **#unknown**
| **#notApplicable**

CallConcurrencyKind ::=
#sequential
| **#guarded**
| **#synchronous**
| **#notApplicable**
| **#unknown**

Multiplicity ::=
LowerMultiplicity
UpperMulitplicity

LowerMultiplicity ::=
integer

UpperMulitplicity ::=
integer
| SimpleName
| /* EMPTY */

BooleanLiteral ::=
#true
| **#false**
| **#unknown**
| **#notApplicable**

Type ::=
#Binding
| **#Refinement**
| **#Usage**
| **#Trace**
| **#Model**
| **#Subsystem**
| **#Package**
| **#Association**
| **#AssociationClass**
| **#Interface**
| **#Class**
| **#Node**

| **#Conponent**
| **#DataType**
| **#Primitive**
| **#Operation**
| **#Enumeration**
| **#Structure**
| **#Method**
| **#Attribute**
| **#unknown**
| **#notApplicable**

Alias ::=
SimpleName

ParameterNumber ::=
integer

ArgumentNumber ::=
integer

Name ::=
SimpleName
| CompoundName

OwnerName ::=
SimpleName
| CompoundName
| **#none**

Namespace ::=
SimpleName
| CompoundName
| **#Primitive**
| **#global**

OwnerNamespace ::=
SimpleName
| CompoundName
| **#Primitive**
| **#global**
| **#none**

Discriminator ::=
StringLiteral
| SimpleName
| **#none**

Stereotype ::=

Name:name
| **#none**
| **#unknown**
| **#notApplicable**

DefaultValue ::=
StringLiteral
| **#none**

InitialValue ::=
StringLiteral
| **#none**

Integer ::=
[0..9]+

Simplename ::=
[a..z A..Z_$] [a..z A..Z 0..9_ - , $ ( ) [ ]]+

CompoundName ::=
SimpleName '.' SimpleName
| CompoundName '.' SimpleName

StringLiteral ::=
'''' [.]+ ''''

Appendix E Data-model description

E.1 Table definitions

The following text is a copy of the SQL file used to set-up the design database. This can be used as a reference to the tables and columns when creating metric queries.

```
CREATE TABLE Association_End
(
        project VARCHAR(64) CONSTRAINT AssociationEnd_column1 NOT NULL,
        version INTEGER CONSTRAINT AssociationEnd_column2 NOT NULL,
        name VARCHAR(128) CONSTRAINT AssociationEnd_column3 NOT NULL,
        namespace VARCHAR(255) CONSTRAINT AssociationEnd_column4 NOT
              NULL,
        stereotype VARCHAR(32),
        aggregation_kind VARCHAR(15),
        changeable VARCHAR(15),
        lower_multiplicity VARCHAR(3),
        upper_multiplicity VARCHAR(3),
        target_scope VARCHAR(15),
        is_navigable CHAR(1),
        is_ordered CHAR(1),
        association_name VARCHAR(128) CONSTRAINT
              AssociationEnd_column13 NOT NULL,
        association_namespace VARCHAR(255) CONSTRAINT
              AssociationEnd_column14 NOT NULL,
        type_name VARCHAR(128) CONSTRAINT AssociationEnd_column15 NOT
              NULL,
        type_namespace VARCHAR(255) CONSTRAINT AssociationEnd_column16
              NOT NULL,
        CONSTRAINT AssociationEnd_primary PRIMARY KEY (project,
              version, name, namespace)
);
CREATE TABLE Attribute
(
        project VARCHAR(64) CONSTRAINT Attribute_column1 NOT NULL,
        version INTEGER CONSTRAINT Attribute_column2 NOT NULL,
        name VARCHAR(128) CONSTRAINT Attribute_column3 NOT NULL,
        namespace VARCHAR(255)CONSTRAINT Attribute_column4 NOT NULL,
        owner_name VARCHAR(128)CONSTRAINT Attribute_column5 NOT NULL,
        owner_namespace VARCHAR(255) CONSTRAINT Attribute_column6 NOT
              NULL,
        stereotype VARCHAR(32),
        changeable VARCHAR(15),
        initial_value VARCHAR(255),
        lower_multiplicty VARCHAR(3),
        upper_multiplicty VARCHAR(3),
        owner_scope VARCHAR(15),
        visibility VARCHAR(15),
        type_name VARCHAR(128),
        type_namespace VARCHAR(255),
        CONSTRAINT Attribute_primary PRIMARY KEY (project, version,
              name, namespace)
);
```

```
CREATE TABLE Dependency
(
      project VARCHAR(64) CONSTRAINT Dependency_column1 NOT NULL,
      version INTEGER CONSTRAINT Dependency_column2  NOT NULL,
      name VARCHAR(255) CONSTRAINT Dependency_column3  NOT NULL,
      namespace VARCHAR(255) CONSTRAINT Dependency_column4  NOT NULL,
      type VARCHAR(32),
      stereotype VARCHAR(32),
      client_name VARCHAR(128) CONSTRAINT Dependency_column7  NOT
            NULL,
      client_namespace VARCHAR(255) CONSTRAINT Dependency_column8
            NOT NULL,
      client_type VARCHAR(32) CONSTRAINT Dependency_column9  NOT
            NULL,
      owner_name VARCHAR(128),
      owner_namespace VARCHAR(255),
      supplier_name VARCHAR(128) CONSTRAINT Dependency_column12  NOT
            NULL,
      supplier_namespace VARCHAR(255) CONSTRAINT Dependency_column13
            NOT NULL,
      supplier_type VARCHAR(32) CONSTRAINT Dependency_column14  NOT
            NULL,
      CONSTRAINT Dependency_primary  PRIMARY KEY (project, version,
            name, namespace)
);

CREATE TABLE Deployment
(
      project VARCHAR(64) CONSTRAINT Deployment_column1  NOT NULL,
      version INTEGER CONSTRAINT Deployment_column2   NOT NULL,
      stereotype VARCHAR(32),
      component_name VARCHAR(128) CONSTRAINT Deployment_column4   NOT
            NULL,
      component_namespace VARCHAR(255) CONSTRAINT Deployment_column5
            NOT NULL,
      node_name VARCHAR(128) CONSTRAINT Deployment_column6   NOT
            NULL,
      node_namespace VARCHAR(255) CONSTRAINT Deployment_column7   NOT
            NULL
);

CREATE TABLE Element_Reference
(
      project VARCHAR(64) CONSTRAINT Element_Reference_column1 NOT
            NULL,
      version INTEGER CONSTRAINT Element_Reference_column2 NOT NULL,
      alias VARCHAR(32) CONSTRAINT Element_Reference_column3 NOT
            NULL,
      stereotype VARCHAR(32),
      visibility VARCHAR(15),
      element_name VARCHAR(128) CONSTRAINT Element_Reference_column6
            NOT NULL,
      element_namespace VARCHAR(255) CONSTRAINT
      Element_Reference_column7 NOT NULL,
      element_type VARCHAR(32) CONSTRAINT Element_Reference_column8
            NOT NULL,
      package_name VARCHAR(128) CONSTRAINT Element_Reference_column9
            NOT NULL,
      package_namespace VARCHAR(255) CONSTRAINT
            Element_Reference_column10 NOT NULL,
      CONSTRAINT Element_Reference_primary PRIMARY KEY (project,
            version, alias, package_name, package_namespace)
);
```

```
CREATE TABLE Generalizable_Element
(
        project VARCHAR(64) CONSTRAINT Generalizable_Element_column1
              NOT NULL,
        version INTEGER CONSTRAINT Generalizable_Element_column2 NOT
              NULL,
        name VARCHAR(128) CONSTRAINT Generalizable_Element_column3 NOT
              NULL,
        namespace VARCHAR(255) CONSTRAINT Generalizable_Element_column4
              NOT NULL,
        type VARCHAR(32) CONSTRAINT Generalizable_Element_column5 NOT
              NULL,
        stereotype VARCHAR(32),
        visibility VARCHAR(15),
        isAbstract CHAR(1),
        isActive CHAR(1),
        isInstantiable CHAR(1),
        isLeaf CHAR(1),
        isPersistent CHAR(1),
        isRoot CHAR(1),
        CONSTRAINT Generalizable_Element_primary PRIMARY KEY (project,
              version, name, namespace)
);

CREATE TABLE Generalization
(
        project VARCHAR(64) CONSTRAINT Generalization_column1 NOT NULL,
        version INTEGER CONSTRAINT Generalization_column2 NOT NULL,
        name VARCHAR(128) CONSTRAINT Generalization_column3 NOT NULL,
        namespace VARCHAR(255) CONSTRAINT Generalization_column4 NOT
              NULL,
        discriminator VARCHAR(32),
        stereotype VARCHAR(32),
        depth INTEGER CONSTRAINT Generalization_column7 NOT NULL,
        subtype_name VARCHAR(128) CONSTRAINT Generalization_column8 NOT
              NULL,
        subtype_namespace VARCHAR(255) CONSTRAINT
              Generalization_column9 NOT NULL,
        subtype_type VARCHAR(32) CONSTRAINT Generalization_column10 NOT
              NULL,
        supertype_name VARCHAR(128) CONSTRAINT Generalization_column11
              NOT NULL,
        supertype_namespace VARCHAR(255) CONSTRAINT
              Generalization_column12 NOT NULL,
        supertype_type VARCHAR(32) CONSTRAINT Generalization_column13
              NOT NULL,
        CONSTRAINT Generalization_primary PRIMARY KEY (project,
              version, name, namespace)
);
```

```
CREATE TABLE Implements
(
      project VARCHAR(64) CONSTRAINT Implements_column1 NOT NULL,
      version INTEGER CONSTRAINT Implements_column2 NOT NULL,
      stereotype VARCHAR(32),
      realization_name VARCHAR(128) CONSTRAINT Implements_column4 NOT
            NULL,
      realization_namespace VARCHAR(255) CONSTRAINT
            Implements_column5 NOT NULL,
      realization_type VARCHAR(32) CONSTRAINT Implements_column6 NOT
            NULL,
      specification_name VARCHAR(128) CONSTRAINT Implements_column7
            NOT NULL,
      specification_namespace VARCHAR(255) CONSTRAINT
            Implements_column8 NOT NULL,
      specification_type VARCHAR(32) CONSTRAINT Implements_column9
            NOT NULL
);

CREATE TABLE Method
(
      project VARCHAR(64) CONSTRAINT Method_column1 NOT NULL,
      version INTEGER CONSTRAINT Method_column2 NOT NULL,
      name VARCHAR(128) CONSTRAINT Method_column3 NOT NULL,
      namespace VARCHAR(255) CONSTRAINT Method_column4 NOT NULL,
      owner_name VARCHAR(128),
      owner_namespace VARCHAR(255),
      type VARCHAR(32),
      stereotype VARCHAR(32),
      concurrency VARCHAR(15),
      owner_scope VARCHAR(15),
      visibility VARCHAR(15),
      is_polymorphic CHAR(1),
      is_query CHAR(1),
      CONSTRAINT Method_primary PRIMARY KEY (project, version, name,
            namespace)
);


CREATE TABLE Parameter
(
      project VARCHAR(64) CONSTRAINT Parameter_column1 NOT NULL,
      version INTEGER CONSTRAINT Parameter_column2 NOT NULL,
      name VARCHAR(128) CONSTRAINT Parameter_column3 NOT NULL,
      namespace VARCHAR(255) CONSTRAINT Parameter_column4 NOT NULL,
      method_name VARCHAR(128) CONSTRAINT Parameter_column5 NOT NULL,
      method_namespace VARCHAR(255) CONSTRAINT Parameter_column6 NOT
            NULL,
      stereotype VARCHAR(32),
      default_value VARCHAR(255),
      kind VARCHAR(15),
      parameter_number INTEGER CONSTRAINT Parameter_column10 NOT
            NULL,
      type_name VARCHAR(128),
      type_namespace VARCHAR(255),
      CONSTRAINT Parameter_primary PRIMARY KEY (project, version,
            namespace, parameter_number)
);
```

```
CREATE TABLE Template_Argument
(
        project VARCHAR(64) CONSTRAINT Template_Argument_column1 NOT
                NULL,
        version INTEGER CONSTRAINT Template_Argument_column2 NOT NULL,
        stereotype VARCHAR(32),
        argument_number INTEGER CONSTRAINT Template_Argument_column4
                NOT NULL,
        binding_name VARCHAR(128) CONSTRAINT Template_Argument_column5
                NOT NULL,
        binding_namespace VARCHAR(255) CONSTRAINT
                Template_Argument_column6 NOT NULL,
        type_name VARCHAR(128) CONSTRAINT Template_Argument_column7 NOT
                NULL,
        type_namespace VARCHAR(255) CONSTRAINT
                Template_Argument_column8 NOT NULL,
        CONSTRAINT Template_Argument_primary PRIMARY KEY (project,
                version, binding_namespace, binding_name,
                argument_number)
);

CREATE TABLE Template_Parameter
(
        project VARCHAR(64) CONSTRAINT Template_Parameter_column1 NOT
                NULL,
        version INTEGER CONSTRAINT Template_Parameter_column2 NOT NULL,
        stereotype VARCHAR(32),
        parameter_number INTEGER CONSTRAINT Template_Parameter_column4
                NOT NULL,
        type_name VARCHAR(128) CONSTRAINT Template_Parameter_column5
                NOT NULL,
        type_namespace VARCHAR(255) CONSTRAINT
                Template_Parameter_column6 NOT NULL,
        template_name VARCHAR(128) CONSTRAINT
                Template_Parameter_column7 NOT NULL,
        template_namespace VARCHAR(255) CONSTRAINT
                Template_Parameter_column8 NOT NULL,
        template_type VARCHAR(32) CONSTRAINT Template_Parameter_column9
                NOT NULL,
        CONSTRAINT Template_Parameter_primary PRIMARY KEY (project,
                version, template_name, template_namespace,
                parameter_number)
);
```

E.2 Permissible values

Although the types of the columns in the various tables can be understood from the table definitions the permissible values for these column cannot. Many of the columns reflect the name of entities as given in the data's original source and any names that are allowable in the source are allowed in the table (subject to column width limitations). Some of the columns, however, have a restricted set of permissible values and this section lists these values.

project - Any string (max. 64 characters)

version - Any positive integer

names - A string not including '.' characters (max. length 128 characters)

namespaces - A string containing one or more dot separated names (max. length 255 characters)

stereotypes - Some of the tables have predefined possible values for this column, others have none. Where there are predefined values these will be listed. The listed sets of values for stereotypes is open to extension by future mappings where this is found useful.

Association_End
aggregation_kind - none, shared, composite
changeable - none, frozen, addOnly
lower_multiplicity - Any string (max 3 characters)
upper_multiplicity - Any string (max 3 characters)
target_scope - classifier, instance
is_navigable - 'T' ,'F' (standing for true or false)
is_ordered -'T' ,'F' (standing for true or false)

Attribute
changeable - none, frozen, addOnly
lower_multiplicty - Any string (max 3 characters)
upper_multiplicty - Any string (max 3 characters)
owner_scope - classifier, instance
visibility - public, protected, private, implementation

Dependency
type - Usage, Refinement, Trace
stereotype - becomes, calls, copy, deletion, derived, friend, import, instance, metaclass, powertype, send, subtraction, uses
client_type - Any concrete meta-type:- Association, AssociationClass, AssociationEnd, Attribute, Class, Component, Constraint, Dependency, Generalization, Interface, Method, Model, Node, Operation, Package, Parameter, Refinement, Subsystem, Usage, Trace
supplier_type - As for client_type

Element_Reference
alias - Any string (max 32 characters)
visibility - public, protected, private, implementation
element_type - Any concrete meta-type:- Association, AssociationClass, AssociationEnd, Attribute, Class, Component, Constraint, Dependency, Generalization, Interface, Method, Model, Node, Operation, Package, Parameter, Refinement, Subsystem, Usage, Trace

Generalizable_Element
type - Association, AssociationClass, Class, Component, DataType, Interface, Model, Node, Package, Subsystem
stereotype - facade, framework, implementationClass, metaclass, powertype, process, stub, system, thread, topLevelPackage, type, utility
visibility - public, protected, private, implementation
isAbstract - 'T' ,'F' (standing for true or false)
isActive - 'T' ,'F' (standing for true or false)
isInstantiable - 'T' ,'F' (standing for true or false)
isLeaf - 'T' ,'F' (standing for true or false)
isPersistent - 'T' ,'F' (standing for true or false)
isRoot - 'T' ,'F' (standing for true or false)

Generalization
discriminator - Any string (max 32 characters)
stereotype - extends, inherits, private, subclass, subtype, uses
depth - any non-negative integer
subtype_type - Association, AssociationClass, Class, Component, DataType, Interface, Model, Node, Package, Subsystem
supertype_type - As for subtype_type

Implements
realization_type - Any concrete meta-type:- Association, AssociationClass, AssociationEnd, Attribute, Class, Component, Constraint, Dependency, Generalization, Interface, Method, Model, Node, Operation, Package, Parameter, Refinement, Subsystem, Usage, Trace
specification_type - As for realization_type

Method
type - Operation, Method
stereotype - create, destroy
concurrency - none, guarded, synchronous
owner_scope - classifier, instance
visibility - public, protected, private, implementation
is_polymorphic - 'T' ,'F' (standing for true or false)
is_query - 'T' ,'F' (standing for true or false)

Parameter
default_value - Any string (max 32 characters)
kind - in, inout, out, return
parameter_number - Any positive integer

Template_Argument
argument_number - Any positive integer

Template_Parameter
parameter_number - Any positive integer
template_type - Any concrete meta-type:- Association, AssociationClass,

AssociationEnd, Attribute, Class, Component, Constraint, Dependency, Generalization, Interface, Method, Model, Node, Operation, Package, Parameter, Refinement, Subsystem, Usage, Trace

Appendix F Source mappings

The mappings described in this section all produce output suitable a input for the 'DesignDatabase' datastore.

F.1 Java source-code mapping

The subsections below give details of the translation between elements of Java source code and the intermediate representation file that is output by the mapping.

F.1.1 Packages

A package in Java is a grouping mechanism for Classes and interfaces. They are mapped as GeneralizableElements.

<GeneralizableElement>
<Name> is set as the unqualified name of the package.
<Namespace> is set as the qualified name of the package to which the declared class belongs. If this is a top-level package it is set to '#global'.
<Type> is set as '#Package'
<Stereotype> is always set as '#none', no stereotypes are used with packages.
<Visbility> is always set as '#notApplicable'.
<IsAbstract> is always set as '#notApplicable'.
<IsActive> is always set as '#notApplicable'.
<IsInstantiable> This tag is only applicable to Subsystems. The distinction between an instantiable subsystem and a uninstantiable subsystem is unclear. Therefore, this tag is always set to '#unknown'
<IsLeaf> is always set as '#notApplicable'
<IsPersistent> is always set as '#notApplicable'.
<IsRoot> is always set as '#notApplicable'
</GeneralizableElement>

F.1.2 Compilation units

A compilation unit is a single Java source file (a file with a *.java* extension). Compilation units are mapped as GeneralizableElements.

<GeneralizableElement>
<Name> is set as the name of the source file.
<Namespace> is set as the qualified name of the package into which the files contents are placed.
<Type> is set as '#Component'
<Stereotype> is always set as 'file'.
<Visbility> is always set as '#notApplicable'.
<IsAbstract> is always set as '#notApplicable'.
<IsActive> is always set as '#notApplicable'.
<IsInstantiable> is always set as '#notApplicable'.
<IsLeaf> is always set as '#notApplicable'.
<IsPersistent> is always set as '#notApplicable'.
<IsRoot> is always set as '#notApplicable'.
</GeneralizableElement>

F.1.3 Classes

A class declaration is mapped as a Generalizable_element. If any of the elements within a class makes use of another type in any way a dependency is created between the class and that type. Any references to methods or attributes from other classes which are made from within static initialises also result in a dependency with the class. In Java any classes without a constructor have a default constructor with no parameters produced for them by the compiler (this is also done by the mapping).

<GeneralizableElement>
<Name> is set as the unqualified name of the class.
<Namespace> is set as the qualified name of the class/package to which the declared class belongs.
<Type> is set as '#Class'
<Stereotype> is always set as #none, no stereotypes are used with classes.
<Visbility> is set as #public if the class is declared with the access modifier 'public'. Similarly for protected and private. If no access modifier is used the visibility is set as #implementation.
<IsAbstract> is set as #true if the 'abstract' modifier is used. It is set as #false otherwise.
<IsActive> is set as #true if the class or any of it's superclasses implements Thread, or if the class or any of it's superclasses implement Runnable. It is set as #false otherwise.
<IsInstantiable> This tag is only applicable to Subsystems. It is therefore always set as #notApplicable for classes.
<IsLeaf> is set as #true if the 'final' modifier is used, #unknown otherwise. There is no other way of knowing from the class itself whether it will be extended.
<IsPersistent> is always set as #unknown. There is no straight forward way to infer whether the object of a class will be persistent.
</GeneralizableElement>

<IsRoot> is set as #false for all classes other than Object.
</GeneralizableElement>

F.1.4 Interfaces

An interface declaration is mapped as a Generalizable_element. If any of the elements within a class makes use of another type in any way a dependency is created between the class and that type.

<GeneralizableElement>
<Name> is set as the unqualified name of the interface.
<Namespace> is set as the qualified name of the class/package to which the declared interface belongs.
<Type> is set as '#Interface'
<Stereotype> is always set as #none, no stereotypes are used with interfaces.
<Visbility> is set as #public if the interface is declared with the access modifier 'public'. Similarly for protected and private. If no access modifier is used the visibility is set as #implementation
<IsAbstract> is always set as #true. Interfaces can never be instantiated.
<IsActive> is set as #true if the interface, or any of the interfaces it implements, implement Runnable. It is set as #false otherwise.
<IsInstantiable> This tag is only applicable to Subsystems. It is therefore always set as #notApplicable for classes.
<IsLeaf> is always set as #unknown. There are no indications from the interface itself whether it will be extended by other interfaces.
<IsPersistent> is always set as #notApplicable. Interfaces can't be instantiated and therefore can't have persistent objects.
<IsRoot> is set as #false if the interface extends any other interface, #true otherwise.
</GeneralizableElement>

F.1.5 Extends

If a class extends another class this is captured as a Generalization. Both direct and indirect extension relationships are included in the mapping, i.e., an the relationship between a class and each of it's ancestors is recorded separately.

<Generalization>
<Name> is set as, qualified name of the extending class + ":extends:" + qualified name of the extended class.
<Namespace> is set as the qualified name of the class/package to which the extending class belongs.

<Discriminator> is set as #none
<Stereotype> is always set as #none
<Depth> This is the number of levels between the ancestor class and the descendent. If the ancestor is the immediate parent of the descendent the depth is 1.
<Subtype>
<Name> The unqualified name of the extending class.
<Namespace>The qualified name of the class/package to which the extending class belongs.
<Type>The unqualified name of the meta-model type to which the extending class belongs, i.e., Class or Interface.
<Supertype>
<Name> The unqualified name of the extended class.
<Namespace>The qualified name of the class/package to which the extended class belongs.
<Type>The name of the meta-model type to which the extended class belongs.

</Generalization>

F.1.6 Implements

If a class implements an interface this is captured as a Implements relation.

<Implements>

<Stereotype> is always set as #none
<Realization>
<Name> The unqualified name of the realising element.
<Namespace>The namespace to which the realising element belongs.
<Type>The name of the meta-model type to which the realising element belongs, i.e., '#Class'.
<Specification>
<Name> The unqualified name of the specifying element.
<Namespace>The qualified name of the class/package to which the specifying element belongs.
<Type>The name of the meta-model type to which the specifying element belongs. i.e.'# Interface'

</Implements>

F.1.7 Fields

A field declaration at the class level is mapped as an Attribute. Field declarations within methods/blocks are not mapped as they considered to be implementation detail (rather than design information). Since the initialisation of non-static fields when they are declared is equivalent to their separate initialisation within each constructor, any references to fields, methods, classes within their initialising expressions is mapped as dependencies within each constructor. As the initialisation is considered to take place within the constructor a dependency between the constructor and the field is also produced.

<Attribute>

<Name> The name of the attribute.
<Namespace> The qualified name of it's class/interface.
<Stereotype> is always set as #none. There are no standard stereotypes for attributes.
<Changeable> is set as #frozen if the final modifier is used, #none otherwise. *It is unclear how* UML's 'addOnly' might be used other than for collections.
<InitialValue> is always set as #none. *Although* some attributes are given initial values their extraction from the source code is not yet *implemented.*
<Multiplicity> is currently always set to 1. Array variables (and possibly other collections) might be mapped as having a multiplicity of 'many' but this is currently not implemented.
<OwnerScope> is set as #classifier if the 'static' modifier is used, #false otherwise
<Visbility> is set as #public if the 'public' access modifier is used. Similarly for protected and private. If no access modifier is used the visibility is set as #implementation
<ElementType>
<Name> The unqualified name of the attribute's type.
<Namespace> The qualified name of the attribute type's containing class/package.

</Attribute>

F.1.8 Methods

This section applies to methods other than constructors. A dependency is produced for the use of any field, method or class used within a methods body. Dependencies are also added between the method and the types of it's parameters (including the return value).

<Method>

<Name> The name of the method.
<Namespace> The qualified name of it's class/interface.
<Type> is set as #Operation if the method is abstract, #Method otherwise.
<Stereotype> is set as #destroy if the method is a finalizer, #none otherwise.
<Concurrency> is set as #guarded if the synchronised modifier is used, #unknown otherwise.
<OwnerScope> is set as #classifier if the 'static' modifier is used, #instance otherwise
<Visbility> is set as #public if the 'public' access modifier is used. Similarly for protected and private. If no access modifier is used the visibility is set as #implementation
<IsPolymorphic> is set as #false if the method is static, #true otherwise.
<IsQuery> In deciding on the value for this tag the standard Java® naming conventions are assumed. It is set to #true if the method name begins 'get' or 'is'; #false is the name starts with 'set'; #unknown otherwise.

</Method>

F.1.9 Constructors

Any constructor which doesn't explicitly call another constructor as it's first action (either this(...) or super(...)) does so implicitly. A dependency is added for this implicit method call. A constructor declaration is mapped as an ordinary method except for the following:

Tag values are as for method declarations except for the following:

<Type> is always set as #Method (you can't have an abstract constructor).
<Stereotype> is always set as #create.
<OwnerScope> is always set as #instance
<IsPolymorphic> is always set as #false.
<IsQuery> is always set as #false.

F.1.10 Parameters

Each formal parameter and return value (except 'void') is mapped as a parameter object.

<Parameter>

<Name> The name of the parameter, or 'return' for return values.
<Namespace> The qualified name of the parameter's method.
<Stereotype> is set as #none.
<DefaultValue> is set as #none (Java does not support default values for parameters).
<ParameterDirectionKind> If the type of the parameter is a basic type is a basic type or a String, or if the parameter is final, the ParameterDirectionKind is '#in'. Return values are mapped as '#return'. Otherwise, '#inout' is used.
<ParameterNumber> return values have the value 0. Other parameters are numbered in an ascending sequence from left to right starting at 1.
<ElementType>
<Name> The unqualified name of the parameter's type.
<Namespace> The qualified name of the parameter type's containing class/package.

</Parameter>

F.1.11 Code Blocks

Within code blocks references to other classes, methods or attributes are mapped as Dependency objects between the referenced object and the owner of the block (either a method or a class).

<Dependency>

<Name> is set as qualified name of the client element + ":dependsOn:" + qualified name of the supplier element .
<Namespace> is set as the client's namespace.
<Type> is set as #Usage.
<Stereotype> is set as 'calls' if both client and supplier are methods, #none otherwise.
<Client> The name of the client element.
<Name> The name of the client element.
<Namespace> The qualified name of it's containing class/interface/package.
<Type> If there client's element contains a <Type> tag the corresponding value is used. Otherwise the meta-model type of the client is used.
<Owner>
<Name> is set as #none.
<Namespace> is set as #none.
<Supplier>
<Name> The name of the supplier element.
<Namespace> The qualified name of it's containing

class/interface/package.

<Type> If there supplier's element contains a <Type> tag the corresponding value is used. Otherwise the meta-model type of the supplier is used.

</Dependency>

F.2 Java byte-code mapping

This mapping uses the java reflection classes to extract the structural details from the '.class' files. This means that the level of detail to which systems can be described is limited to level of detail available form the reflection classes. The major problem caused by this is that the reflection classes do not provide any information about method internal structure. This means that metrics requiring method-call or attribute-use information will not work on projects using this mapping.

F.3 Rational Rose mapping

This mapping uses the Roses built-in scripting language to extract the design information contained within Roses internal meta-model and output the information in DDL format. The mapping consists of around one thousand lines of BASIC script. It can be installed within the Rose tool to create an additional menu item which, when selected, creates the DDL equivalent of the current model. This mapping can only be run from within Rose and the script rerun after any changes to the model to keep the DDL script up-to-date.

Appendix G Questionnaire

Blank

In

Original

OOSE assignment debriefing

Answers given on this questionnaire will not affect the marking of the assignment in any way. Your name is necessary only for triangulating the data sources.

Name: _____

"Pre-Assignment" Questions

This set of questions is intended to get an idea of the your experience and knowledge in relevant areas prior to the start assignment.

Background and experience

How much programming experience did you have *before* the OOSE unit? And in what languages?

Only fill in details for those languages you have used. You should complete the duration for the highest level of experience. Use the blank lines for any languages not listed.

	Duration (in months)					Type of experience		
Programming language	<1	<3	<6	≤12	>12	casual	formal course	professional use
Java								
C++								
C								
Pascal								
BASIC								
COBOL								
FORTRAN								

How much experience did you have with object-oriented design *before* the OOSE unit?

Only fill in details for those methods you have used. You should complete the duration for the highest level of experience. Use the blank lines for any methods not listed.

	Duration (in months)					Type of experience		
Method	<1	<3	<6	≤12	>12	casual	formal course	professional use
UML								
OMT								
Booch								
Coad & Yourdon								
Shlaer / Mellor								
SA/SD								
SSADM								

Understanding of the case study

In the following questions circle the answer which best represents your level of agreement with the given statement.

I had a clear understanding of the *purpose* of the Vocabulary Tutor program

agree strongly agree neutral disagree disagree strongly

I had a clear understanding of the *high-level design* of the Vocabulary Tutor program

agree strongly agree neutral disagree disagree strongly

I had a clear understanding of the *source code* of the Vocabulary Tutor program

agree strongly agree neutral disagree disagree strongly

"Doing-Assignment" Questions

This set of questions is intended to get an idea of how you went about doing the assignment.

Time and effort spent on doing the assignment

In the following questions circle the closest value.

Estimate (in hours) the amount of time you spent on background research for this assignment

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 >15

Estimate the amount of time you spent analysing the measures

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 >15

Estimate the amount of time you spent writing-up your assignment

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 >15

Metric selection

How did you choose the metrics on which to base your analysis?

_____

_____

_____

_____

In the following two questions circle one answer

Did you consider taking a pick'n'mix approach to selecting metrics? (taking some metrics from one authors set and some from another)

YES NO UNSURE

Did you consider using a different set of metrics for analysing the design refinement than you did for the initial assessment?

YES NO UNSURE

Use of data sources

How important were the following data-sources/activities in forming your comments on the designs?

Give separate answers for their importance in the initial formation of ideas and their importance in providing sufficient corroboration of an idea to convince you of that idea's validity. Use the blank lines to add any other data-sources/activities which contributed to the formation of your comments.

1 - not important ⟶ very important - 4

	initial formation of ideas				corroboration of ideas			
Data-source / Activity	1	2	3	4	1	2	3	4
measurement results								
source code								
design diagrams								
CRC card enactment								

Opinions on individual metrics

Grade the following list of metrics according to how understandable they were and how useful you found them for your assignment. If you didn't research or use a particular metric (or can't recall any details) just leave it blank.

1 - difficult to understand ⟶ easy to understand - 4
1 - not useful at all ⟶ very useful - 4

	Understandable					Useful				
Metric name	No opinion	1	2	3	4	No opinion	1	2	3	4
Chidamber & Kemerer										
Weighted methods per class										
Response set for a class										
Coupling between object classes										
Lack of cohesion in methods										
Depth of inheritance tree										
Number of children										
Abreu (MOOD)										
Attribute hiding factor										
Method hiding factor										
Attribute inheritance factor										
Method inheritance factor										
Coupling factor										
polymorphism factor										
Lorenz & Kidd										
Number of subsystems										
Number of message sends										
Average method size										
Number of public instance methods										
Number of instance methods										
Average number of instance method										
Number of instance variables										
Average number of instance variabl										
Number of class methods										
Number of class variables										
Class hierarchy nesting level										
Number of abstract classes										
Use of multiple inheritance										
Number of methods overridden										
Number of methods inherited										
Number of methods added										
Specialization index										
Average number of parameters method										
Class coupling										
Number of methods thrown away										
Number of classes thrown away										

Tool use

In the second part of the assignment how many different versions of the project (not including the original) did you measure using the tool?

Circle one answer

0 1 2 3 4 5 >6

Did the usability/functionality of the tool adversely affect the way went about the process of the assignment?

Circle one answer

YES NO UNSURE

How might the usability and functionality of the tool be improved?

_____

_____

_____

_____

_____

_____

_____

"Post-Assignment" Questions

This set of questions is intended capture your reflections on the process of the assignment and to get an idea of how your knowledge and views on related topics have changed as a result of doing the assignment.

Reflections on the process

Which aspects of the assignment did you find particularly difficult?

_____

_____

_____

_____

What could have been changed to make the assessment/refinement easier?

What additional information or resources would have been useful?

Changes in views and knowledge

In the following questions circle one answer

Do you feel you that relating ideas about quality to measurement data has helped you understand design quality more fully?

YES NO UNSURE

Given suitable tool support, would you choose to apply measurement to help you design future projects?

YES NO UNSURE

Would your experience with this assignment make you more supportive of a suggestion to implement a measurement programme in a company for which you worked?

YES NO UNSURE

The End (Thank you very much)

Appendix H Design documents

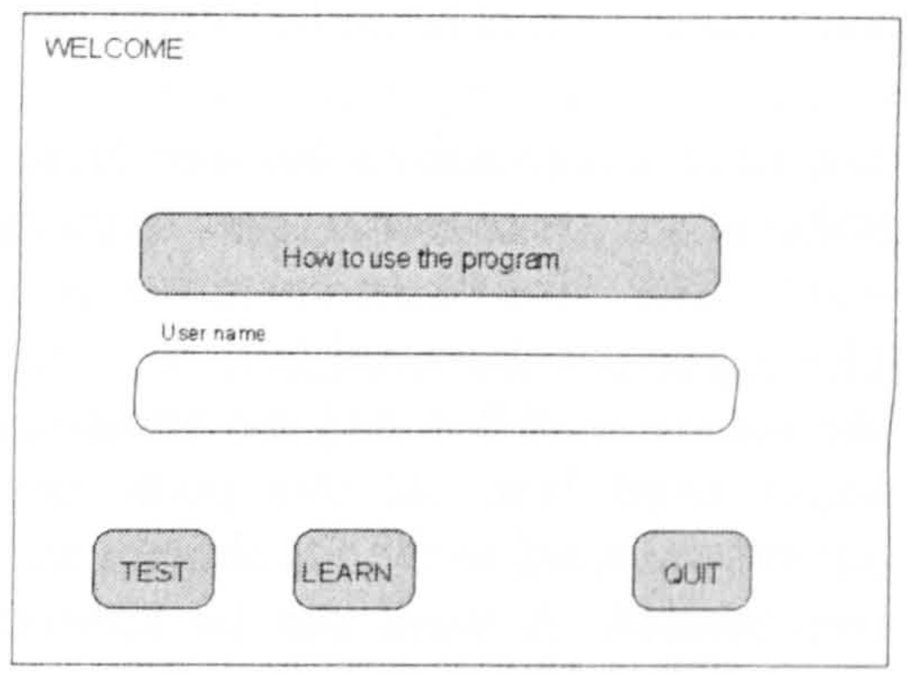
Blank

In

Original

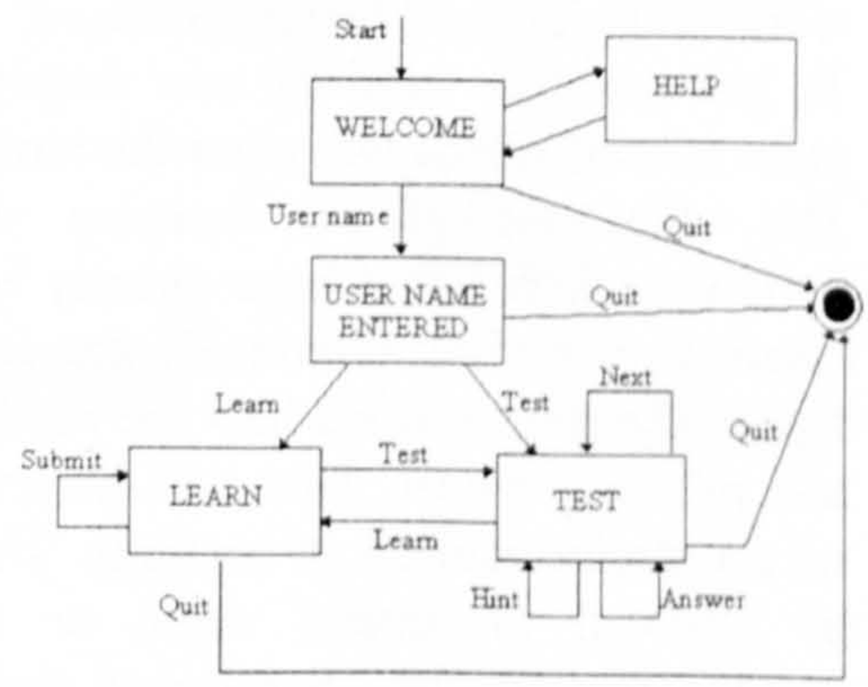
An Informal Specification

2nd Language Vocabulary Tutor



This is an informal specification of a small program which we will use in our design review exercise. The program is aimed at those users whose first language is English and who are learning a second language. The purpose of the program is to aid in the tedious task of learning vocabulary in the second language. Research suggests that the most successful tactic for learning vocabulary in a second, and any subsequent language, is to force the user to actively construct a range of links. These links connect words and ideas from the first, or source language, to words in the second, or target language. This program embodies that principle. Every word that the user attempts to learn has to be entered into the program and a collection of links have to be created for each word. There is no built in dictionary, the user does the work and this constructive effort aids the learning process. When the user wishes to test his ability to remember the words he has entered he may ask the program to test him. If he fails to remember accurately and requests some help, then he is given help in the form of a hint; a hint is a reminder of a link that he originally entered. One implication of this model of learning is that any user will need a copy of the program and a vocabulary of their own. The vocabulary will be kept as a separate file and will grow over time.

The 3 screen pictures in this note use a simple convention for the various screen elements. Push buttons are shown shaded. Boxes which are editable are shown with a thin line surround and boxes for system output are shown with a thicker line surround.

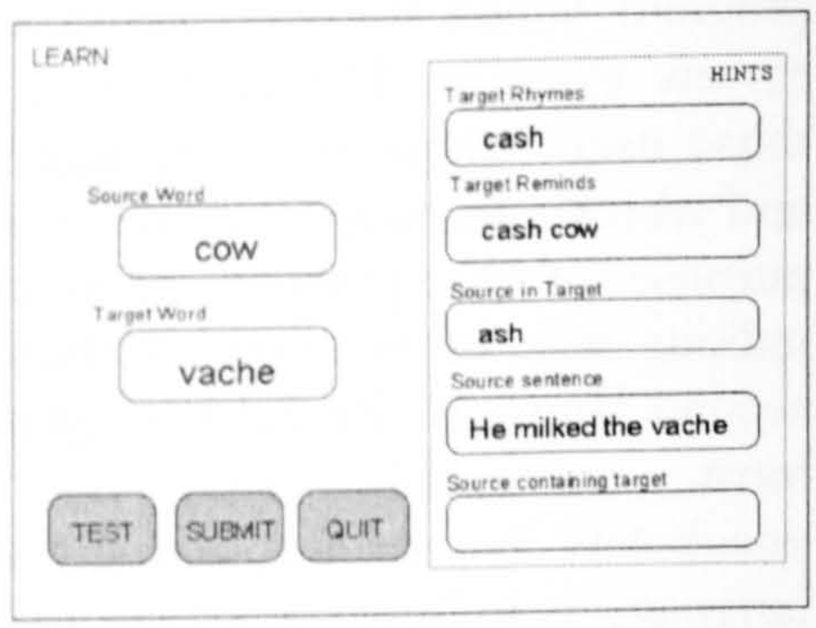


The program consists of 4 screens; a welcome screen, a help screen, a learn screen and a test screen. The program commences with a welcome screen. The user now has the opportunity to go to the help screen to read the help information which describes how to use the program. Each user of the system will need their own vocabulary file so at this point the program needs to either retrieve or create the file. Users are required to type their name in the user name box and the system will either create a new vocabulary file for them (if it is their first time) or open their existing vocabulary file. The user may now select to learn some more words or to be tested on the words they already have in their vocabulary file. Pressing the appropriate button will take them to the appropriate screen. They may also quit straight out of the program.

The state transition diagram shows the various paths that a user can take through the program. This shows that the welcome screen is quite complex; it has modal behaviour. A user cannot jump straight to the learning or testing screens unless they have already loaded (or created) their vocabulary file. It should be noted that this is not intended to be an exemplary user interface. Our focus is on the internal design of

the program, we accept that the program is slightly user-hostile in places.

We explain the behaviour of the learn screen next as, on first use, the user has to enter some words before a test becomes meaningful.



The learn screen allows the user to perform two tasks; adding new words and reviewing existing words. The basic learn operation is as follows. The user types the word they wish to learn into the source word box and the translation into the target word box. At this point the user can submit the word to the vocabulary and it will be remembered. A word can be submitted to the vocabulary without any hints, but it must have a translation. The next part is the most important as far as the learning goes. The user has now to create a series of hints and type them into the appropriate boxes. The range of possible hints are outlined below.

Target rhymes - find a word from the users source language that rhymes with the target word.

Target reminds - find a word from the users source language which somehow reminds the user of the target word.

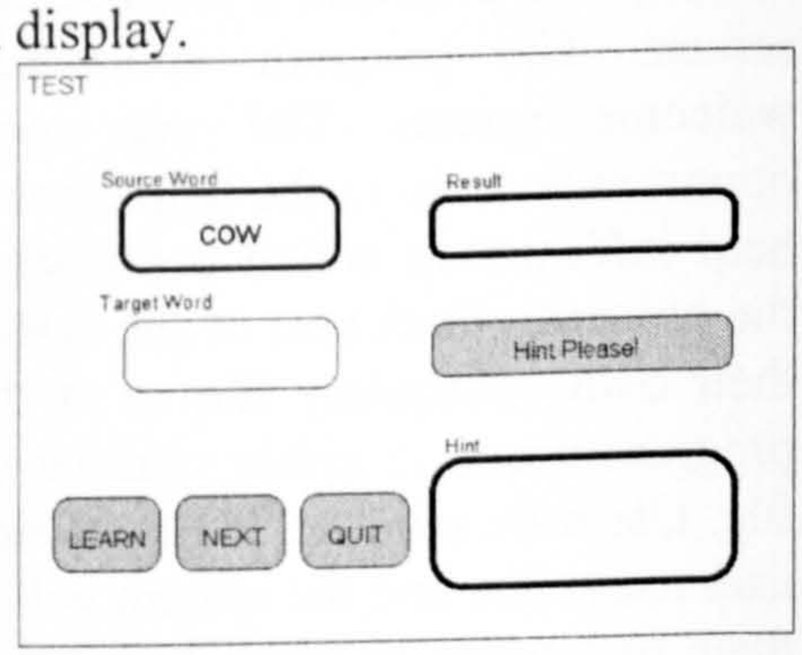
Source in target - find a word from the users source language which appears as a part of the target word. This can either be a spelling or a similar pronunciation.

Source sentence - construct a short sentence in the users source language with the source word replaced by the target word.

Source containing target - find a source word which contains the target word as a component. Again, this can either be by spelling or pronunciation.

When the user is happy with his hints he may press submit and the source word, target word, and hints will be remembered.

To learn another word the user simply has to type in a new source and target word and press submit which will clear the hints currently on display.



The other use of the learning screen is in reviewing words that have already been entered. Users may review previously entered words by typing the source word into the source word box and pressing return. This will retrieve the word, its translation and all of its hints. They will now be allowed to amend some or all of their previously entry for the retrieved word. Again, in order to ensure that the changes are remembered the user must press submit. At any point in the learn screen the user may either quit the program or enter the test screen.

The final screen is the test screen. The user is presented with a random word from their vocabulary file and asked for the translation. They type the translation into the target word window and upon pressing the return key the word is checked. They either get the answer correct and receive a positive message in the result window or they get it wrong and then get a conciliatory message in the result window. If they wish to try again they may do so. They may also ask for a hint. If they ask for a hint then a hint

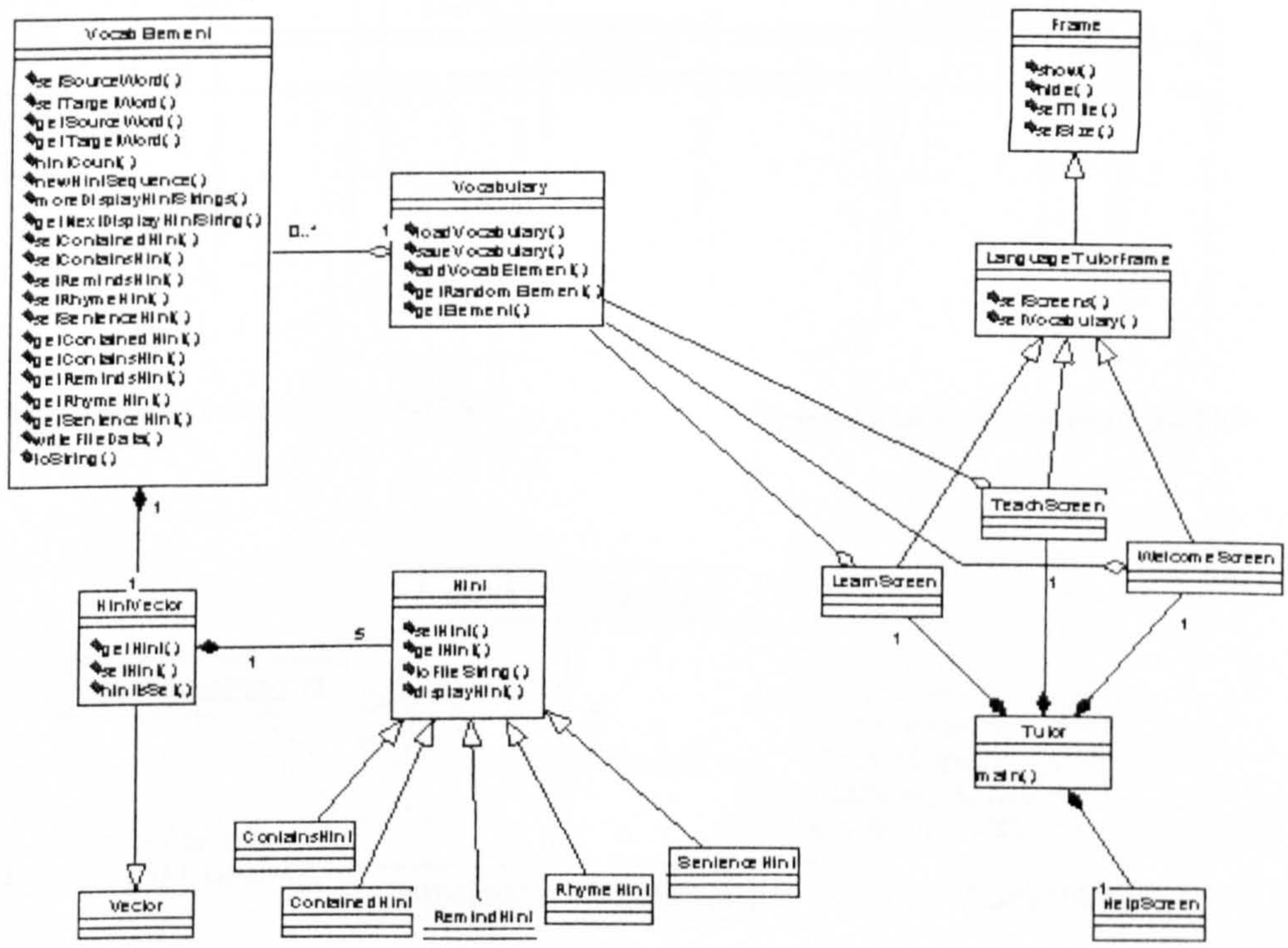
selected at random from those connected with the word will appear in the hint window and this may be enough to prompt a retry. They may keep asking for hints until they have seen all of the hints for that word. If they keep asking they are told there are no more hints and given the correct answer.

If the user wishes to move on to another word then they *simply press the next button* and are given the next random word. This continues until all of the words have been used. At any point in the test screen the user may either quit the program or enter the learn screen.

Note that when the source sentence hint is being used in the test screen the target word will need to be replaced with the correct source word otherwise the answer will be given away.

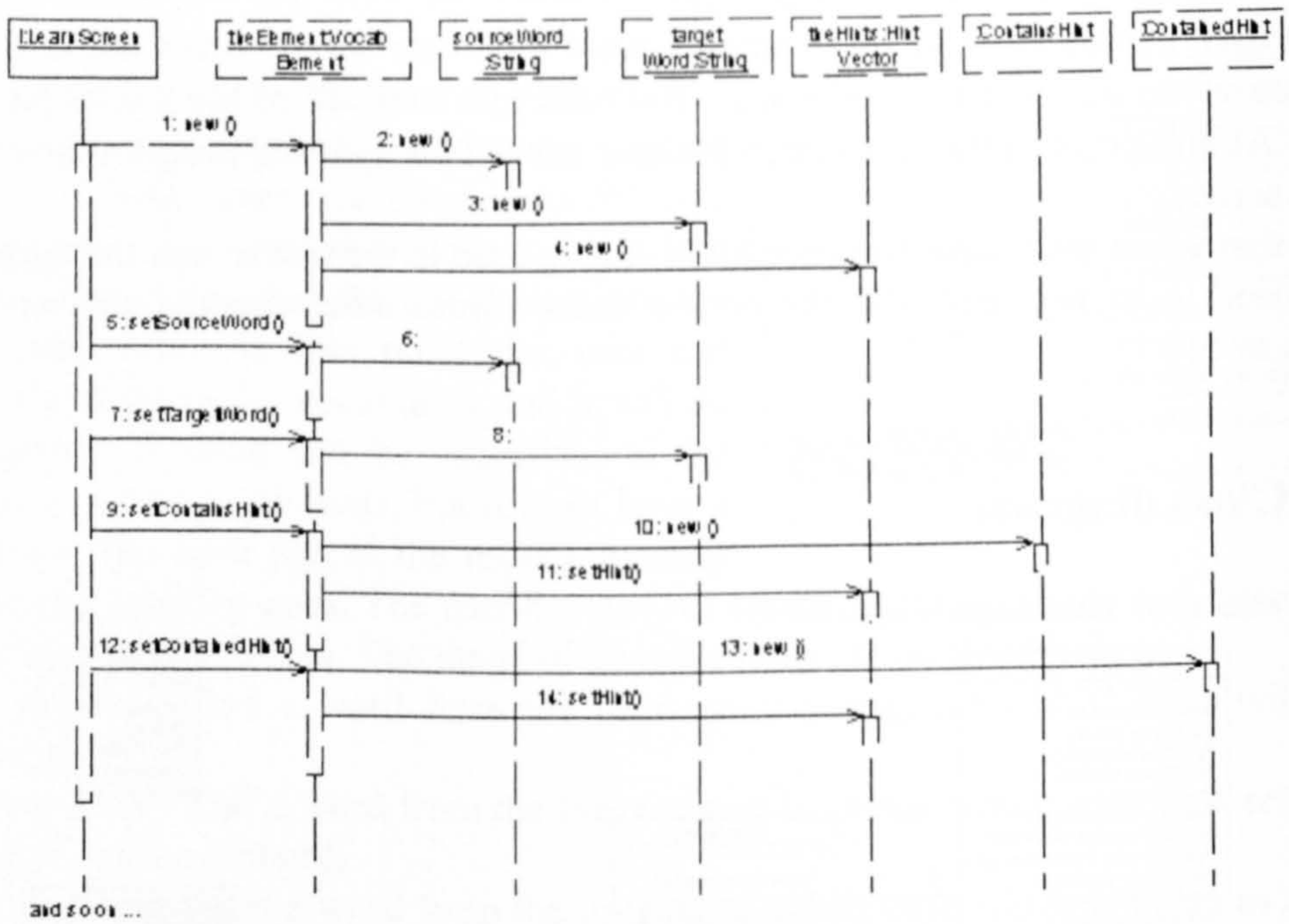
H.1 Class diagram

Vocab tutor class diagram details

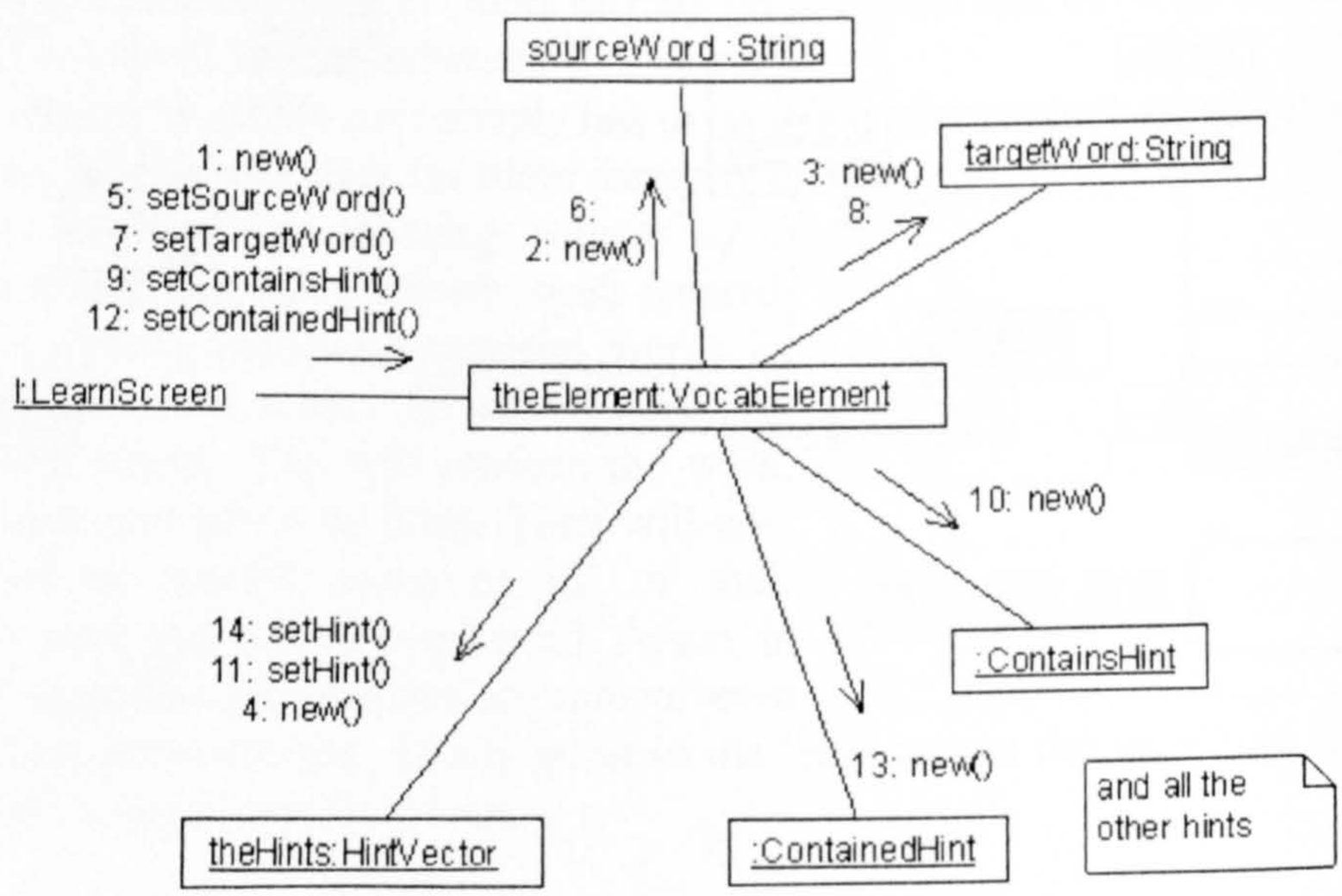


H.2 Behavioural diagrams

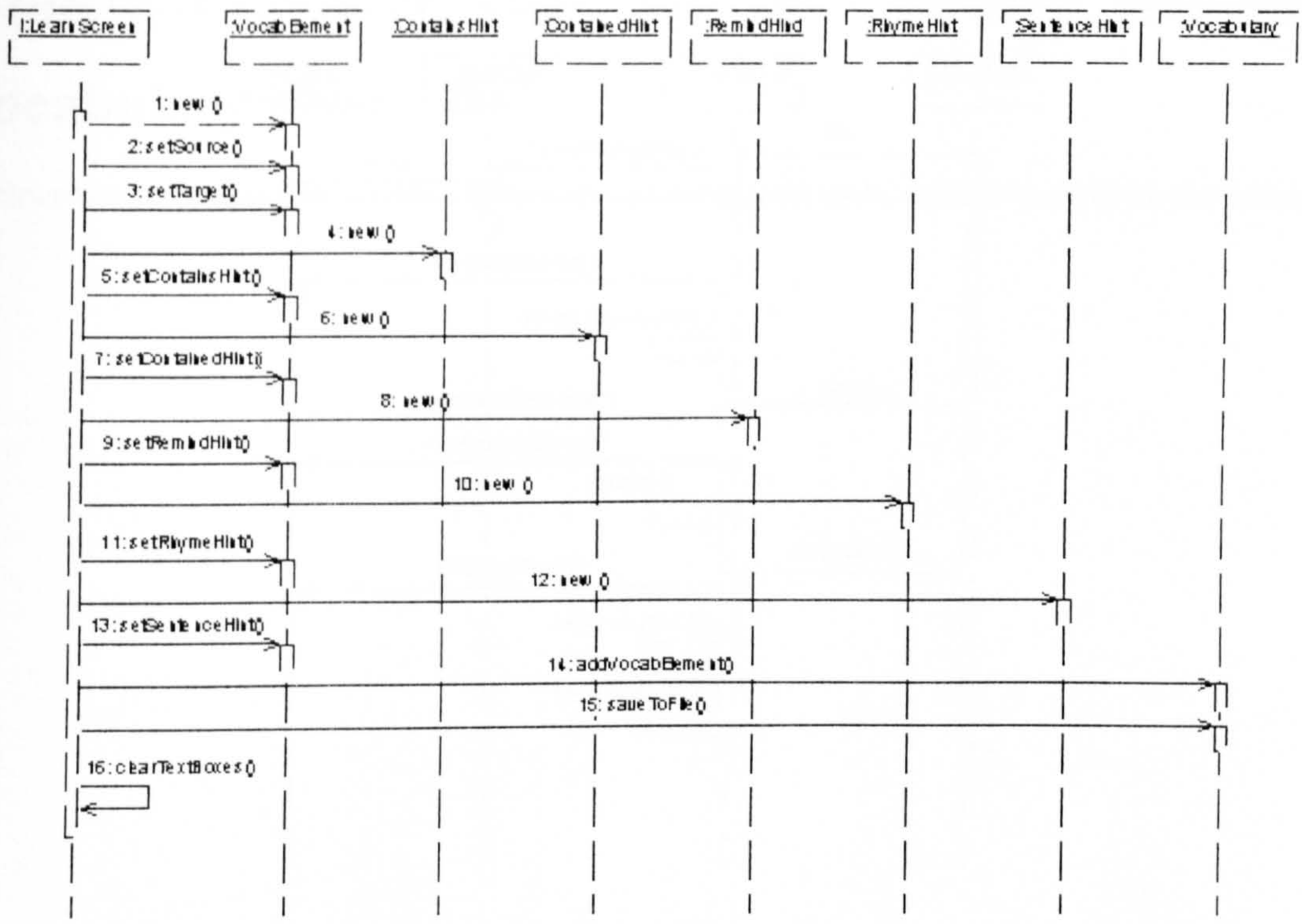
Adding a new vocab element - detail



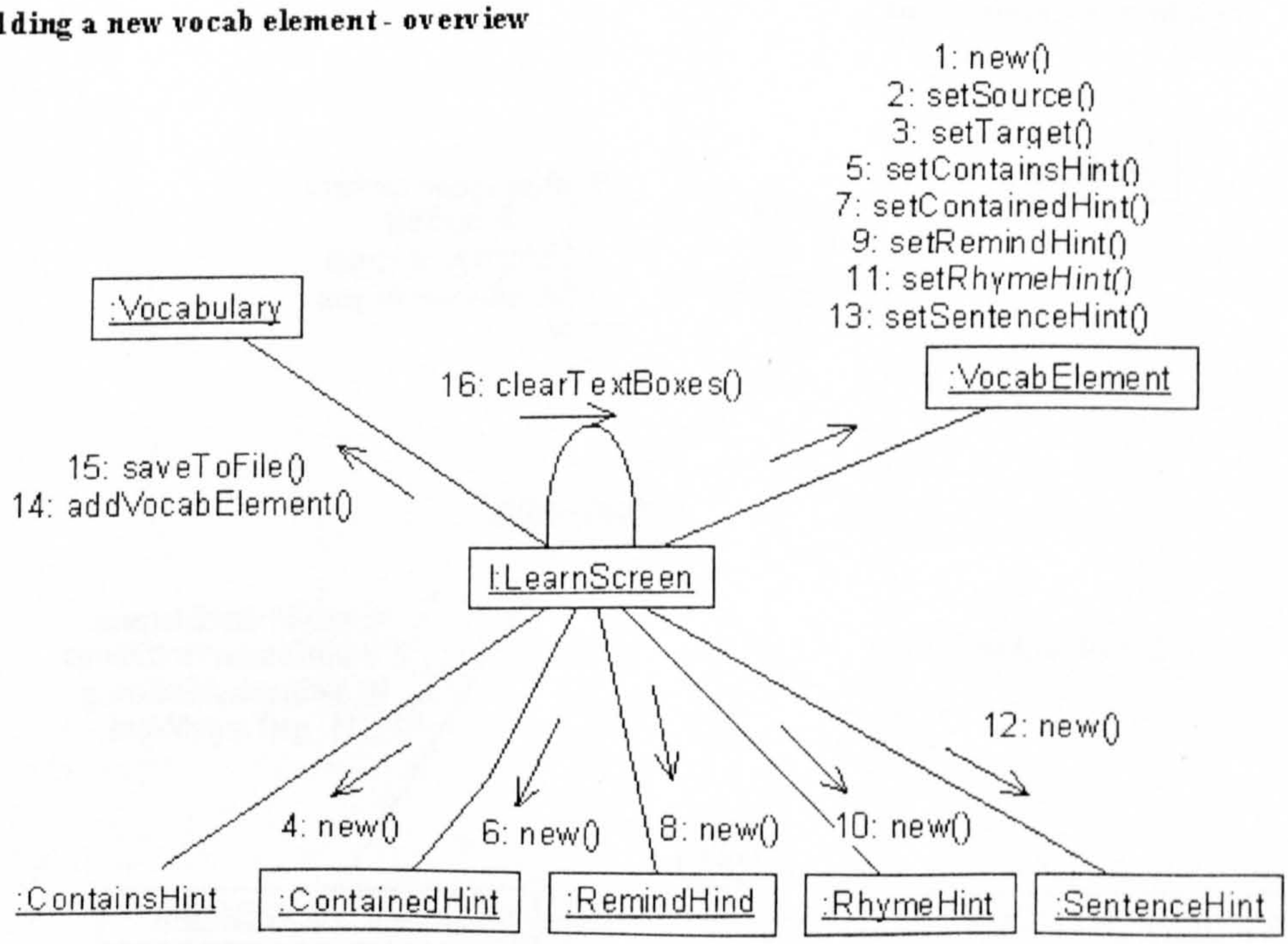
Adding a new vocab element -detail



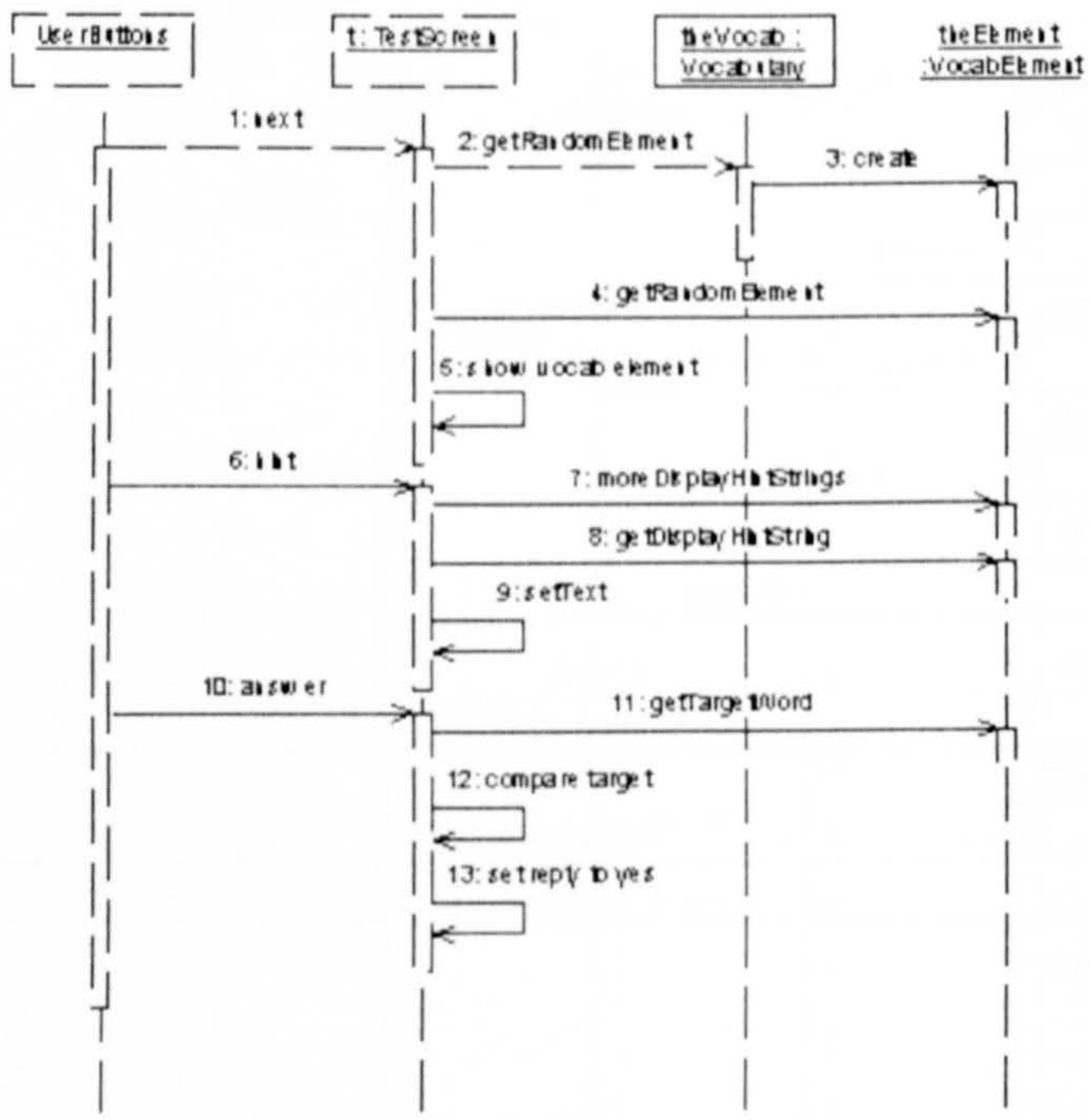
Adding a new vocab element - overview



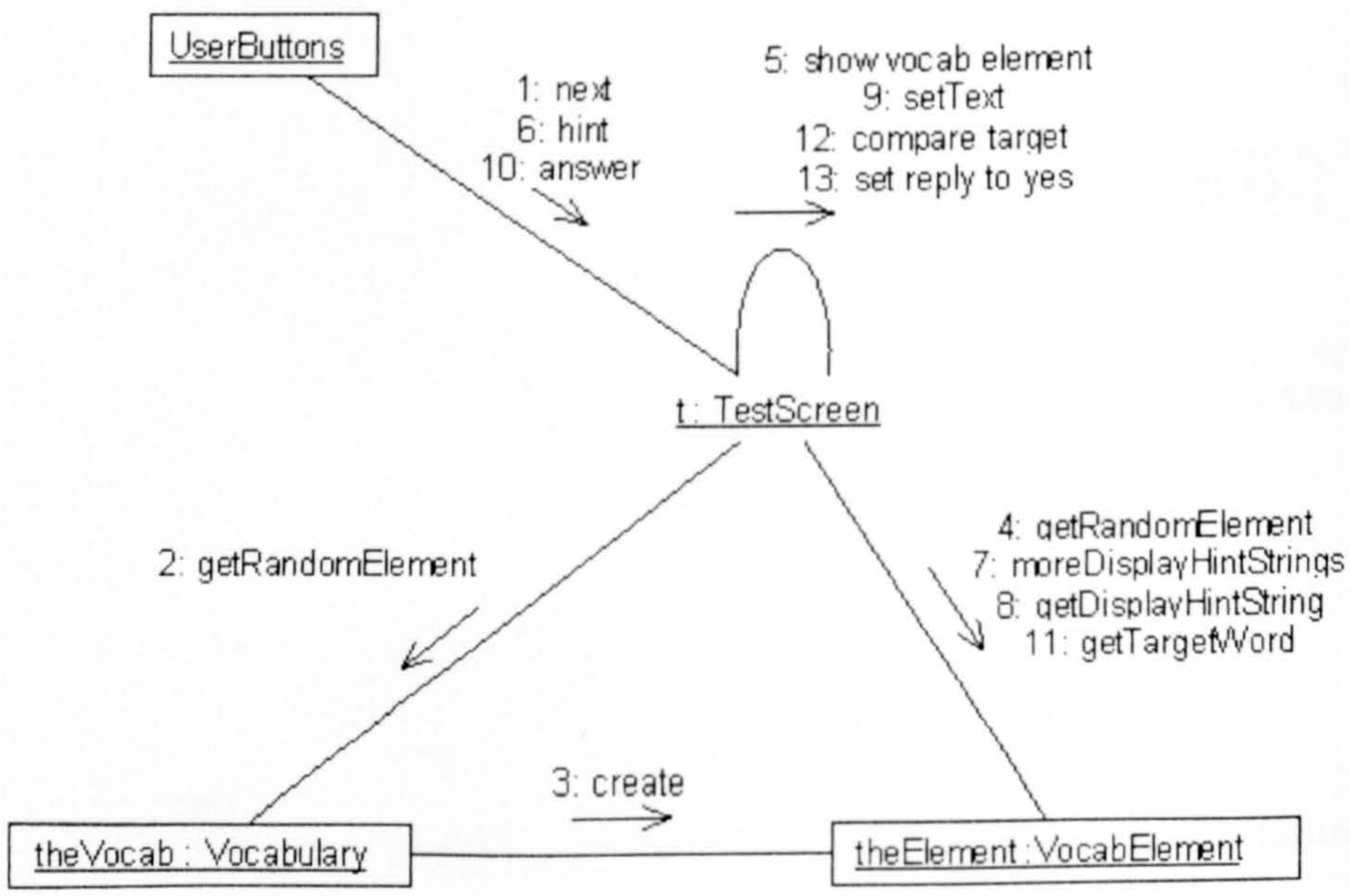
Adding a new vocab element - overview



Test screen - using a hint



Test Screen - using a hint



Appendix I Source code for the 'Second language vocabulary tutor'

Blank

In

Original

```
/*
File Name: ContainedHint.java
Author: S.Webster
Modified: 14/11/98
*/

package vocabtutoroose;

class ContainedHint extends Hint
{

     public ContainedHint ()
       {
            super();
     } // end ContainedHint


     public ContainedHint (String s, String t, String h)
     {
     super(s, t, h);
     } // end ContainedHint


    public ContainedHint(Hint h)
     {
            super(h);
     } // end ContainedHint



     public String displayHint()
     {
            // return a version for display
            return new String (hintString);
        } // end displayHint
}
```

```
/*
File Name: ContainsHint.java
Author: S.Webster
Modified: 14/11/98
*/

package vocabtutoroose;

class ContainsHint extends Hint
{

     public ContainsHint ()
      {
            super();
      } // end ContainsHint


      public ContainsHint (String s, String t, String h)
     {
            super(s, t, h);
      } // end ContainsHint


      public ContainsHint(Hint h)
     {
            super(h);
      } // end ContainsHint


      public String displayHint()
     {
       // return a version for display
            return new String (hintString);
      } // end displayHint
}
```

```
/*
File Name: HelpScreen.java
Author: S.Webster
Modified: 14/11/98
*/

package vocabtutoroose;

import java.awt.*;
import java.awt.event.*;

public class HelpScreen extends Frame implements WindowListener
{

     TextArea                instructionsTA                =                new
     TextArea("",10,30,TextArea.SCROLLBARS_NONE);
       String words = "Welcome to the second language tutor\n\n" +
       "You need to type your name to open your vocabulary file\n\n"
     + "Remember to press return when you enter a new word";

    public HelpScreen()
    {
        super();
        setSize(300,210);
        add(instructionsTA);
        instructionsTA.setText(words);

        addWindowListener(this);
    }
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
    public void windowClosing(WindowEvent e)
    {
        this.hide();
    }
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}

}
```

```
/*
File Name: Hint.java
Author: S.Webster
Modified: 14/11/98
*/
package vocabtutoroose;

public abstract class Hint {

    // Fields
    protected String hintString = null;
    protected String sourceString = null;
    protected String targetString = null;

    protected static final String DELIMITER = "|";

    //Methods

    public Hint()
      {
            sourceString = null;
            targetString = null;
            hintString = null;
    } // end Hint

    public Hint(String s, String t, String h)
      {
            sourceString = new String(s);
            targetString = new String(t);
            hintString = new String(h);
    } // end Hint

    public Hint(Hint h)
      {
        if (h != null)
        {
            sourceString = new String(h.sourceString);
            targetString = new String(h.targetString);
            hintString = new String(h.hintString);
            }
    } // end Hint

    public void setHint (String s)
      {
        hintString = new String(s);
    } // end setHint


    public String getHint()
      {
        return new String (hintString);
    } // end getHint

    public String toFileString()
    {
        return  new String( ((sourceString == null)? "":
              sourceString) + DELIMITER + ((targetString == null)
              ?"":targetString) + DELIMITER + (((hintString ==
              null)||(hintString.equals(""))))?"*":hintString ));
    } // end toFileString


    public abstract String displayHint();

} // end Hint
```

Blank

In

Original

```
/*
File Name: HintVector.java
Author: S.Webster
Modified: 14/11/98
*/

package vocabtutoroose;

import java.util.Vector;

public class HintVector extends Vector
{
    public static final int CONTAINS = 1;
    public static final int CONTAINED = 2;
    public static final int REMINDS = 3;
    public static final int RHYMES = 4;
    public static final int SENTENCE = 5;

    public HintVector()
    {
        super(6);
        setSize(6);
        for(int i = 1; i <= 4; i++)
        {
            setHint(new ContainsHint("", "", ""), i);
             // ugly fix as Hint is abstract
        }
        setHint(new SentenceHint(), HintVector.SENTENCE);
    }

    public HintVector(HintVector hv)
    {
        super(6);
        setSize(6);
        for(int i = 1; i <= 4; i++)
        {
            setHint(new ContainsHint(hv.getHint(i)), i);
        }
        setHint(new SentenceHint(hv.getHint(HintVector.SENTENCE)),
                                                HintVector.SENTENCE);
    }

    public Hint getHint(int hintPosition)
    {
        return (Hint)elementAt(hintPosition);
    }

    public void setHint(Hint theHint, int hintPosition)
    {
        setElementAt(theHint, hintPosition);
    }

    public boolean hintIsSet(int hintPosition)
    {
        return !(getHint(hintPosition).getHint().equals(""));
    }
}
```

```
/*
File Name: LanguageTutorFrame.java
Author: S.Webster
Modified: 14/11/98
*/

package vocabtutoroose;

import java.awt.*;
import java.awt.event.*;

// Provides the ability to close

public abstract class LanguageTutorFrame extends Frame
{

    Vocabulary theVocab;

    public LanguageTutorFrame()
    {
        setSize(300, 200);
        setTitle(getClass().getName());
    }

    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowOpened(WindowEvent e) {}
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}


    public abstract void setScreens(LanguageTutorFrame first,
                                    LanguageTutorFrame second);

    public abstract void setVocabulary(Vocabulary theVocab);

} // end LanguageTutorFrame
```

```
/*
File Name: LearnScreen.java
Author: S.Webster
Modified: 14/11/98
*/

package vocabtutoroose;

import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class LearnScreen extends LanguageTutorFrame implements
ActionListener, WindowListener
{

    TextField rhymeTF = new TextField(10);
    TextField remindTF = new TextField(10);
    TextField containsTF = new TextField(10);
    TextField sentenceTF = new TextField(10);
    TextField containedTF = new TextField(10);

    Label rhymeL = new Label("rhyme");
    Label remindL = new Label("remind");
    Label containsL = new Label("contains");
    Label sentenceL = new Label("sentence");
    Label containedL = new Label("contained");

    Label sourceL = new Label("source");
    Label targetL = new Label("target");


    TextField sourceTF = new TextField(10);
    TextField targetTF = new TextField(10);

    Button quitButton = new Button("Quit");
    Button submitButton = new Button("Submit");
    Button testButton = new Button("Test");

    LanguageTutorFrame learnScreenL;
    LanguageTutorFrame testScreenL;

    VocabElement theElement;
    Vocabulary theVocab;

    LearnScreen()
    {

        super();

        setSize(600, 420);
        setLayout(null);

        add(sourceL);
        sourceL.setBounds(40,60,100,20);

        add(sourceTF);
        sourceTF.setBounds(40,80,210,40);

        add(targetL);
        targetL.setBounds(40,160,100,20);

        add(targetTF);
        targetTF.setBounds(40,180,210,40);
```

```
        add(quitButton);
        quitButton.setBounds(20,320,60,60);

        add(submitButton);
        submitButton.setBounds(120,320,60,60);

        add(testButton);
        testButton.setBounds(220,320,60,60);

        add(rhymeL);
        rhymeL.setBounds(300,20,100,20);

        add(rhymeTF);
        rhymeTF.setBounds(300,40,260,40);

        add(remindL);
        remindL.setBounds(300,100,100,20);

        add(remindTF);
        remindTF.setBounds(300,120,260,40);

        add(containsL);
        containsL.setBounds(300,180,100,20);

        add(containsTF);
        containsTF.setBounds(300,200,260,40);

        add(sentenceL);
        sentenceL.setBounds(300,260,100,20);

        add(sentenceTF);
        sentenceTF.setBounds(300,280,260,40);

        add(containedL);
        containedL.setBounds(300,340,100,20);

        add(containedTF);
        containedTF.setBounds(300,360,260,40);

        quitButton.addActionListener(this);
        submitButton.addActionListener(this);
        testButton.addActionListener(this);

        rhymeTF.addActionListener(this);
        remindTF.addActionListener(this);
        containsTF.addActionListener(this);
        sentenceTF.addActionListener(this);
        containedTF.addActionListener(this);

        sourceTF.addActionListener(this);
        targetTF.addActionListener(this);

            addWindowListener(this);

    } // end constructor

    public void actionPerformed(ActionEvent evt)
    {
        String next = evt.getActionCommand();

        //user pushes the Quit button
        if (next.equals("Quit"))
        {
            System.exit(0);
```

```
        }

        //user pushes the Submit button
        if (next.equals("Submit"))
        {
            String s = sourceTF.getText();
            String t = targetTF.getText();

            theElement = new VocabElement();

            if( (s != null && !s.equals("")) &&
                (t != null && !t.equals("")) )
            {
    // there must be both a source and target field as a minimum
    // construct a new or revised vocab element and save it

                theElement.setSourceWord(s);
                theElement.setTargetWord(t);

                theElement.setContainsHint(new ContainsHint(s, t,
                                           containsTF.getText()));
                theElement.setContainedHint(new ContainedHint(s,
                                        t, containedTF.getText()));
                theElement.setRemindsHint(new RemindHint(s, t,
                                             remindTF.getText()));
                theElement.setRhymeHint(new RhymeHint(s, t,
                                             rhymeTF.getText()));
                theElement.setSentenceHint(new SentenceHint(s, t,
                                           sentenceTF.getText()));

                theVocab.addVocabElement(theElement);

                // clear all the text fields
                sourceTF.setText("");
                targetTF.setText("");
                containsTF.setText("");
                containedTF.setText("");
                remindTF.setText("");
                rhymeTF.setText("");
                sentenceTF.setText("");

                //then save to file
                try
                {
                    theVocab.saveVocabulary();
                }
                catch (IOException e)
                {
                    System.exit(1);
                    e.toString();
                }

            } // end if - both a source and target string

        } // end if submit

        // user pushes the Test button
        if (next.equals("Test"))
        {
            this.hide();
            testScreenL.show();
        }

        // user presses return in the source TextField
        if (evt.getSource() == sourceTF)
```

```
{
    // check if there is a VocabElement for this source
    if(theVocab.hasElement(sourceTF.getText()))
    {
        // if there is then load it
        theElement = theVocab.getElement(
                                    sourceTF.getText());

        // and set the text fields
        targetTF.setText(theElement.getTargetWord());
        // if the hint is null, display an empty string
        if(theElement.getContainsHint() != null)
        {
            containsTF.setText(theElement.
            getContainsHint().getHint());
        }

        if(theElement.getContainedHint() != null)
        {
            containedTF.setText(theElement.
            getContainedHint().getHint());
        }

        if(theElement.getRemindsHint() != null)
        {
            remindTF.setText(theElement.getRemindsHint().g
            etHint());
        }

        if(theElement.getRhymeHint() != null)
        {
            rhymeTF.setText(theElement.getRhymeHint().
            getHint());
        }

        if(theElement.getSentenceHint() != null)
        {
            sentenceTF.setText(theElement.
            getSentenceHint().getHint());
        }

    }
    // if there is not then create a new one
    else
    {
        theElement = new VocabElement();
        // and set the text fields to show empty
        targetTF.setText("");
        containsTF.setText("");
        containedTF.setText("");
        remindTF.setText("");
        rhymeTF.setText("");
        sentenceTF.setText("");

    }
}

} // end actionPerformed


public void setScreens(LanguageTutorFrame learn,
                       LanguageTutorFrame test)
{
    learnScreenL = learn;
```

```
        testScreenL = test;
    }

    public void setVocabulary(Vocabulary theVocab)
    {
        this.theVocab = theVocab;
    }

} // end LearnScreen
```

```
/*
File Name: RemindHint.java
Author: S.Webster
Modified: 14/11/98
*/

package vocabtutoroose;

public class RemindHint extends Hint {


      public RemindHint ()
      {
            super();
      } // end RemindHint


      public RemindHint (String s, String t, String h)
      {
            super(s, t, h);
      } // end RemindHint


      public RemindHint(Hint h)
      {
            super(h);
      } // end RemindHint



public String displayHint()
  {
      // return a version for display
      return new String (hintString);
  } // end displayHint


} // end RemindHint
```

```
/*
File Name: RhymeHint.java
Author: S.Webster
Modified: 14/11/98
*/

package vocabtutoroose;

public class RhymeHint extends Hint {


      public RhymeHint ()
      {
            super();
      } // end RhymeHint


      public RhymeHint (String s, String t, String h)
      {
         super(s, t, h);
      } // end RhymeHint


      public RhymeHint(Hint h)
      {
         super(h);
      } // end RhymeHint



    public String displayHint()
      {
          // return a version for display
          return new String (hintString);
      } // end displayHint


} // end RhymeHint
```

```
/*
File Name: SentenceHint.java
Author: S.Webster
Modified: 14/11/98
*/

package vocabtutoroose;

public class SentenceHint extends Hint {


      public SentenceHint ()
      {
            super();
      } // end SentenceHint


      public SentenceHint (String s, String t, String h)
      {
            super(s, t, h);
      } // end SentenceHint


      public SentenceHint(Hint h)
      {
         super(h);
      } // end SentenceHint


   public String displayHint()
      {
            // if there is a version of the word in the string
            if(hintString.indexOf(targetString) != -1)
            {
                  int targetWordStart = hintString.IndexOf
                                              (targetString);
             int targetWordEnd = targetWordStart +
                                        targetString.length();
                  String firstPart, thirdPart;

                  if (targetWordStart == 0)
                  {
                        firstPart = "";
                  }
                  else
                  {
                        firstPart = hintString.substring(0,
                                        targetWordStart);//-1
                  }

                  if (targetWordEnd > hintString.length())
                  {
                        thirdPart = "";
                  }
                  else
                  {
                        thirdPart                                    -
hintString.substring(targetWordEnd);
                  }

                  return      firstPart + sourceString + thirdPart;
            }
            else
            {
```

```
                return "malformed sentence hint";
            }

        } // end displayHint

    } // end SentenceHint
```

```
/*
File Name: TestScreen.java
Author: S.Webster
Modified 14/11/98
*/


package vocabtutoroose;

import java.awt.*;
import java.awt.event.*;

public class TestScreen extends LanguageTutorFrame implements
ActionListener, WindowListener
{
    TextField resultTF = new TextField(10);
    TextArea hintTA = new TextArea("",2,10,
                                        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(10);
    TextField targetTF = new TextField(10);

    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;

    VocabElement theElement;

    boolean firstMessage = true; // allows alternate conciliatory
                                 // message displays

    public TestScreen()
    {
        super();

        setSize(600, 420);
        setLayout(null);

        add(sourceL);
        sourceL.setBounds(40,60,100,20);

        add(sourceTF);
        sourceTF.setBounds(40,80,210,40);

        add(targetL);
        targetL.setBounds(40,160,100,20);

        add(targetTF);
        targetTF.setBounds(40,180,210,40);


        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);
        resultL.setBounds(300,60,100,20);

        add(resultTF);
        resultTF.setBounds(300,80,260,40);

        add(hintButton);
        hintButton.setBounds(300,180,260,40);


        add(hintL);
        hintL.setBounds(300,300,100,20);

        add(hintTA);
        hintTA.setBounds(300,320,260,60);


        quitButton.addActionListener(this);
        learnButton.addActionListener(this);
        nextButton.addActionListener(this);

        targetTF.addActionListener(this);
        hintButton.addActionListener(this);

        addWindowListener(this);

    }

    public void actionPerformed(ActionEvent evt)
    {
        String next = evt.getActionCommand();
        if (next.equals("Quit"))
        {
            System.exit(0);
        }
        if (next.equals("Learn"))
        {
            this.hide();
            learnScreenT.show();
        }
        if (next.equals("Next"))
        {
            // retrieve a new VocabElement from the Vocabulary
            theElement = theVocab.getRandomElement();
            theElement.newHintSequence();
            // display the VocabElement
            sourceTF.setText(theElement.getSourceWord());
            // clear all other text fields
            hintTA.setText("");
            targetTF.setText("");
            resultTF.setText("");
        }
        if (next.equals("Hint Please!"))
        {
            // check that there is a current element
            if(theElement != null)
```

```
                {
                    // check that there are hints to give
                    if(theElement.moreDisplayHintStrings())
                    {
                          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()))
                {
                    //say well done
                    resultTF.setText("Well done, that is correct!");
                }
                else
                {
                    //say 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 setScreens(LanguageTutorFrame learn,
                                        LanguageTutorFrame test)
        {
            learnScreenT = learn;
            testScreenT = test;
        }

        public void setVocabulary(Vocabulary theVocab)
        {
            this.theVocab = theVocab;
        }

    } // end TestScreen
```

```
/*
Driver program for Vocab Tutor

File Name: Tutor.java
Author: S.Webster
Modified: 14/11/98
*/

package vocabtutoroose;

public class Tutor {

    static WelcomeScreen w;
    static LearnScreen l;
    static TestScreen t;

    public static void main (String args[])
      {
        w = new WelcomeScreen();
        l = new LearnScreen();
        t = new TestScreen();

        w.setScreens(l, t);
        l.setScreens(l, t);
        t.setScreens(l, t);

        w.show();

    } // end main

} // end Tutor
```

```
/*
File Name: VocabElement.java
Author: S.Webster
Modified: 14/11/98

A vocabulary element is a collection of related information about a
single
language element that the user is attempting to learn.
It contains the source word, the target word and a collection of
hints.
The user may request the hints to be put into a random sequence and
may
then call for the display strings from each hint in the sequence.
*/

package vocabtutoroose;

import java.util.*;
import java.io.*;

public class VocabElement {

    // Fields
    private String sourceWord;
    private String targetWord;

    private HintVector theHints;

     // These variables are not part of the objects state they are //
     transient
    private Hint[] theTempHints;
    private int hintCursor;

    private static final String DELIMITER = "|";
         // used for file handling


    //Methods

    public VocabElement()
     {
            sourceWord = new String("");
            targetWord = new String("");

            theHints = new HintVector();
      }

    public VocabElement(String s, String t)
      {
          //create internal objects
            sourceWord = new String(s);
            targetWord = new String(t);

            theHints = new HintVector();
    }

    public VocabElement(VocabElement v)
      {
          //create internal objects
            sourceWord = new String(v.getSourceWord());
            targetWord = new String(v.getTargetWord());

            theHints = new HintVector(v.getHintVector());
    } // end copy constructor
```

```
public VocabElement(String fileString)
{
        // create tokeniser
        StringTokenizer t = new StringTokenizer(fileString, "|");

        //create internal objects
        sourceWord = new String(t.nextToken());
        targetWord = new String(t.nextToken());

        theHints = new HintVector();

        //read in the first possible hint
        String source1 = t.nextToken();
        String target1 = t.nextToken();
        String hint1 = t.nextToken();

        // check it is a valid hint
        if (hint1.equals("*"))
        {
              theHints.setHint(new ContainsHint(sourceWord,
                      targetWord, ""), HintVector.CONTAINS);
        }
        else
        {
              theHints.setHint(new ContainsHint(source1, target1,
                                  hint1), HintVector.CONTAINS);
        }

        //read in the second possible hint
        String source2 = t.nextToken();
        String target2 = t.nextToken();
        String hint2 = t.nextToken();

        // check it is a valid hint
        if (hint2.equals("*"))
        {
              theHints.setHint(new ContainedHint(sourceWord,
                      targetWord, ""), HintVector.CONTAINED);
        }
        else
        {
              theHints.setHint(new          ContainedHint(source2,
target2,
                                  hint2), HintVector.CONTAINED);
        }

        //read in the third possible hint
        String source3 = t.nextToken();
        String target3 = t.nextToken();
        String hint3 = t.nextToken();
        // check it is a valid hint
        if (hint3.equals("*"))
        {
               theHints.setHint(new RemindHint(sourceWord,
                       targetWord, ""), HintVector.REMINDS);
        }
        else
        {
              theHints.setHint(new RemindHint(source3, target3,
                                  hint3), HintVector.REMINDS);
        }

    //read in the fourth possible hint
        String source4 = t.nextToken();
```

```
            String target4 = t.nextToken();
            String hint4 = t.nextToken();

            // check it is a valid hint
            if (hint4.equals("*"))
            {
                  theHints.setHint(new RhymeHint(sourceWord,
                           targetWord, ""), HintVector.RHYMES);
            }
            else
            {
                  theHints.setHint(new RhymeHint(source4, target4,
                                    hint4), HintVector.RHYMES);
            }

        //read in the fifth possible hint
            String source5 = t.nextToken();
            String target5 = t.nextToken();
            String hint5 = t.nextToken();

            // check it is a valid hint
            if (hint5.equals("*"))
            {
                  theHints.setHint(new SentenceHint(sourceWord,
                           targetWord, ""), HintVector.SENTENCE);
            }
            else
            {
                  theHints.setHint(new SentenceHint(source5, target5,
                                    hint5), HintVector.SENTENCE);
            }
      } // end constructor from file string


      public void setSourceWord (String s)
      {
            sourceWord = s;
      } // end setSourceWord


      public void setTargetWord (String s)
    {
            targetWord = s;
      } // end setTargetWord


      public String getSourceWord()
      {
            return sourceWord;
      } // end getSourceWord


      public String getTargetWord()
      {
        return targetWord;
      } // end getTargetWord

    public HintVector getHintVector()
    {
        return theHints;
    }

/* ----------------------------------------------------*/
```

```
public void newHintSequence()
{

    hintCursor = -1; // reset to start of array
    int hintCount = 0;

    // initialise the array
    theTempHints = new Hint[6];
     // need an empty location at the end


    // put in all the items
    if (theHints.hintIsSet(HintVector.CONTAINED))
    {
        theTempHints[++hintCursor] = new ContainedHint(
                    theHints.getHint(HintVector.CONTAINED));
        hintCount++;
    }
    if (theHints.hintIsSet(HintVector.CONTAINS))
    {
        theTempHints[++hintCursor] = new ContainsHint(
                    theHints.getHint(HintVector.CONTAINS));
        hintCount++;
    }
    if (theHints.hintIsSet(HintVector.REMINDS))
    {
        theTempHints[++hintCursor] = new RemindHint(theHints.
                                getHint(HintVector.REMINDS));
        hintCount++;
    }
    if (theHints.hintIsSet(HintVector.RHYMES))
    {
        theTempHints[++hintCursor] = new RhymeHint(theHints.
                                getHint(HintVector.RHYMES));
        hintCount++;
    }
    if (theHints.hintIsSet(HintVector.SENTENCE))
    {
        theTempHints[++hintCursor] = new SentenceHint(
                    theHints.getHint(HintVector.SENTENCE));
        hintCount++;
    }

        // shuffle them
        // for 1 to hintCount loop
        // put this one into a temp - put a random one here - put
        //temp where the random came from

    for(int i = 0; i < hintCount; i++)
    {
        int j = (int)(Math.random() * hintCount);
        Hint tempHint = theTempHints[i];
        theTempHints[i] = theTempHints[j];
        theTempHints[j] = tempHint;
    }

    hintCursor = -1; // reset to start of array

} // end newHintSequence


public boolean moreDisplayHintStrings()
{
        return (theTempHints[hintCursor + 1] != null) ;
} // end moreDisplayHintStrings
```

```
        public String getNextDisplayHintString()
        {
            return new String(theTempHints[++hintCursor]
                                            .displayHint());
        } // end getNextHint

/* ----------------------------------------------------*/

        public void setContainedHint(Hint h)
        {
              theHints.setHint(h, HintVector.CONTAINED);
        } // end setContainedHint


        public void setContainsHint(Hint h)
        {
              theHints.setHint(h, HintVector.CONTAINS);
        } // end setContainsHint


        public void setRemindsHint(Hint h)
        {
              theHints.setHint(h, HintVector.REMINDS);
        } // end setRemindsHint


        public void setRhymeHint(Hint h)
        {
              theHints.setHint(h, HintVector.RHYMES);
        } // end setRhymeHint


        public void setSentenceHint(Hint h)
        {
              theHints.setHint(h, HintVector.SENTENCE);
        } // end setSentenceHint

/* ----------------------------------------------------*/

        public Hint getContainedHint()
        {
              return theHints.getHint(HintVector.CONTAINED);
        } // end getContainedHint


        public Hint getContainsHint()
        {
              return theHints.getHint(HintVector.CONTAINS);
        } // end getContainsHint


        public Hint getRemindsHint()
        {
              return theHints.getHint(HintVector.REMINDS);
        } // end getRemindsHint
```

```
        public Hint getRhymeHint()
        {
              return theHints.getHint(HintVector.RHYMES);
        } // end getRhymeHint


        public Hint getSentenceHint()
        {
              return theHints.getHint(HintVector.SENTENCE);
        } // end getSentenceHint


      public void writeFileData(PrintWriter outFile) throws

            IOException
            {
                  outFile.print(((sourceWord == null)?"":sourceWord) +
                                                          DELIMITER);
                  outFile.print(((targetWord == null)?"":targetWord) +
                                                          DELIMITER);

              if (theHints.getHint(HintVector.CONTAINED).equals(""))
              {
                  ContainedHint tempContainedHint = new ContainedHint(
                                                      sourceWord,
      targetWord, "*");
                  outFile.print(tempContainedHint.toFileString() +
                                                          DELIMITER);
              }
              else
              {
                   outFile.print((theHints.getHint(HintVector.CONTAINED
                                  )).toFileString() + DELIMITER);
              }

              if (theHints.getHint(HintVector.CONTAINS).equals(""))
              {
                  ContainsHint tempContainsHint = new ContainsHint(
                                  sourceWord, targetWord, "*");
                  outFile.print(tempContainsHint.toFileString() +
                                                          DELIMITER);
              }
              else
              {
                   outFile.print((theHints.getHint(HintVector.CONTAINS)
                                  ).toFileString() + DELIMITER);
              }

              if (theHints.getHint(HintVector.REMINDS).equals(""))
              {
                       RemindHint tempRemindHint = new RemindHint(
                                  sourceWord, targetWord, "*");
                  outFile.print(tempRemindHint.toFileString() +
                                                          DELIMITER);
              }
              else
              {
                   outFile.print((theHints.getHint(HintVector.REMINDS))
                                  .toFileString() + DELIMITER);
              }

              if (theHints.getHint(HintVector.RHYMES).equals(""))
              {
                       RhymeHint tempRhymeHint = new RhymeHint(sourceWord,
                                                      targetWord, "*");
```

```
                outFile.print(tempRhymeHint.toFileString() +
                                                DELIMITER);
            }
            else
            {
                outFile.print((theHints.getHint(HintVector.RHYMES)).
                                    toFileString() + DELIMITER);
            }

            if (theHints.getHint(HintVector.SENTENCE).equals(""))
            {
                SentenceHint tempSentenceHint = new SentenceHint(
                                    sourceWord, targetWord, "*");

                outFile.print(tempSentenceHint.toFileString()+"\n");
                }
                else
                {
                    outFile.print((theHints.getHint(HintVector.SENTENCE)
                                                ).toFileString() + "\n");
                }
        }       // end      writeFileData

        public String toString()
        {
            return (sourceWord + " " + targetWord);
        }
    } // end VocabElement
```

```
/*
File Name: Vocabulary.java
Author: S.Webster
Modified: 14/11/98
*/

package vocabtutoroose;

import java.util.*;
import java.io.*;


public class Vocabulary
{

        // Fields
        private final int MAX_WORDS = 20;
        String ownerName;
        int wordCount;
        VocabElement[] words;

        //Methods
        public Vocabulary(String name)
        {
                ownerName = name;
                wordCount = 0;
                words = new VocabElement[MAX_WORDS];
        }

        public void loadVocabulary(String fileName)
        {
                try
                {
                        File f = new File(fileName);
                        if(!f.exists())
                        {
                        PrintWriter pw = new PrintWriter(new
                                                        FileWriter(f));
                        pw.print(ownerName + "\n");
                        pw.print(wordCount + "\n");
                        pw.close();
                        }
                // turn vocab file into something reasonable to read from
                        BufferedReader vocabFile = new BufferedReader(new
                                                        FileReader(f));
                        ownerName = new String(vocabFile.readLine());
                        wordCount = Integer.valueOf(vocabFile.readLine())
                                                        .intValue();
                        for(int i = 0; i < wordCount; i++)
                        {
                                                        words[i]=new
                VocabElement(vocabFile.readLine());
                        }
                        vocabFile.close();
                }
                catch (FileNotFoundException e)
                {
                        System.out.println("loadVocabulary Error: " + e);
                        System.exit(1);
                }
                catch (IOException e)
                {
                        System.out.println("loadVocabulary Error: " + e);
                        System.exit(1);
                }
```

```
} // end loadVocabulary

public void saveVocabulary() throws IOException
{
      try
      {
            PrintWriter pW = new PrintWriter(new FileWriter(
                                    ownerName + ".dat"));

            pW.print(ownerName + "\n");
            pW.print(wordCount + "\n");
            for(int i = 0; i < wordCount; i++)
            {
                  words[i].writeFileData(pW);
            }
            pW.close();
      }
      catch (IOException e)
      {
            System.out.println("Error: " + e);
            System.exit(1);
      }

} // end saveVocabulary


public void addVocabElement(VocabElement ve)
{
      // check to see if we already have this element
      if(hasElement(ve.getSourceWord()))
      {
            // we do so override the old element
            for(int i = 0; i < wordCount; i++)
            {

            if(words[i].getSourceWord().equals(ve.
                                    getSourceWord()))
                  {
                        words[i] = new VocabElement(ve);
                  }
            }
      }
      else
      {
            words[wordCount++] = new VocabElement(ve);
      }
}     // end addVocabElement


public VocabElement getRandomElement()
{
      return words[(int)(Math.random() * (wordCount))];

} // end getRandomElement

public boolean hasElement(String sourceWord)
{
      // linear search the array, return true if found
      boolean itemFound = false;
      for(int i = 0; i < wordCount; i++)
      {
            if(words[i].getSourceWord().equals(sourceWord))
            {
                  itemFound = true;
```

```
                }
            }
            return itemFound;
        }  // end hasElement

        public VocabElement getElement(String sourceWord)
        {
            // linear search the array, return element, exception if
        //not found
            VocabElement sendBack = new VocabElement();

            if (this.hasElement(sourceWord))
            {
                for(int i = 0; i < wordCount; i++)
                {

            if(words[i].getSourceWord().equals(sourceWord))
                        {
                            sendBack = words[i];
                        }
                }
            }
            else
            {
            System.out.println("Vocabulary.getElement() " +
                                        " item not present");
                System.exit(1);
            }

            return sendBack;
        } // end getElement

    } // end Vocabulary
```

```
/*
File Name: WelcomeScreen.java
Author: S.Webster
Modified: 14/11/98
*/

package vocabtutoroose;

import java.awt.event.*;
import java.awt.*;
import java.io.*;

public class WelcomeScreen extends LanguageTutorFrame implements
ActionListener, WindowListener
{
      Button quitButton = new Button("Quit");
      Button learnButton = new Button("Learn");
      Button testButton = new Button("Test");
      Button helpButton = new Button("How to use this program");
      TextField userInputTF = new TextField(30);

      LanguageTutorFrame learnScreenW;
      LanguageTutorFrame testScreenW;

      HelpScreen help = new HelpScreen();

      Vocabulary theVocab;

      WelcomeScreen()
      {

            super();
            setSize(600,420);
            setLayout(null);

            add(helpButton);
            helpButton.setBounds(140, 100, 260, 40);

            add(userInputTF);
            userInputTF.setBounds(140, 200, 260, 40);

            add(learnButton);
            learnButton.setBounds(60, 320, 60, 60);

            add(testButton);
            testButton.setBounds(180, 320, 60, 60);

            add(quitButton);
            quitButton.setBounds(400, 320, 60, 60);

            helpButton.addActionListener(this);
            userInputTF.addActionListener(this);

            quitButton.addActionListener(this);
            learnButton.addActionListener(this);
            testButton.addActionListener(this);

            addWindowListener(this);

      }  // end constructor


      public void actionPerformed(ActionEvent evt)
      {
```

```
                String next = evt.getActionCommand();
                if (next.equals("Quit"))
                {
                    System.exit(0);
                }
                if (next.equals("Learn"))
                {
                      // if there is something in the input field
                      if (!userInputTF.getText().equals(""))
                      {
                      theVocab = new Vocabulary(userInputTF.getText()
                                                          .trim());

                      theVocab.loadVocabulary(userInputTF.getText().
                                                  trim() + ".dat");
                      learnScreenW.setVocabulary(theVocab);
                      testScreenW.setVocabulary(theVocab);
                      this.hide();
                      learnScreenW.show();
                      }
                }
                if (next.equals("Test"))
                {
                      // if there is something in the input field
                      if (!userInputTF.getText().equals(""))
                      {
                      theVocab = new Vocabulary(userInputTF.getText()
                                                          .trim());

                      theVocab.loadVocabulary(userInputTF.getText().
                                                  trim() + ".dat");
                      learnScreenW.setVocabulary(theVocab);
                      testScreenW.setVocabulary(theVocab);
                      this.hide();
                      testScreenW.show();
                      }
                }
                if (next.equals("How to use this program"))
                {
                      //call help screen
                      help.show();
                }

          }  // end actionPerformed


          public void setScreens(LanguageTutorFrame learn,
                                          LanguageTutorFrame test)
          {
                learnScreenW = learn;
                testScreenW = test;
          }

          public void setVocabulary(Vocabulary theVocab)
          {
                // set in the instance of this class so
                // no need to implement anything here
          }


     } // end WelcomeScreen
```

Appendix J Case study task

Blank

In

Original

BSc Software Engineering Management Year 4 (1998-99)
Object Oriented Software Engineering - Assignment

Set date: 24/11/98
Due date: 17/02/99
Return date: 10/03/99

Introduction
This assignment is concerned with OO design evaluation. You are given a complete working (small) application. You are first to assess the quality of the design of the application and you will be expected to employ a design evaluation tool to help in this process. You will then amend the design and implementation of the product, without altering its behaviour, and then assess the quality of the finished result again employing the design evaluation tool.

The process
The assignment is in 2 parts, a preliminary report to be completed and handed in late January and a final report to be handed in on the due date mid February.
The preliminary report is to contain your initial analysis of the design. Your analysis should indicate the weak points in the design and should back up those arguments with heuristics and metrics.
The final report should outline the design changes you have made and explain the motivation for these changes, again you should back up these arguments with heuristics and metrics. You should also critically evaluate the design/evaluation process you followed.
To help in the construction of your preliminary report the final lab session before the Christmas break will be given over to using the design evaluation tool.
It will be possible to make the tool available in other labs after Christmas to allow you some extra flexibility.

The case study
The case study we are using is the Second Language Vocabulary Tutor. This is a small application to allow learners of a second language to increase their vocabulary through a process of actively learning the new words and testing themselves on their progress. The details of the case study are to be found on the OOSE web pages. The source code for the case study will be available on the web page and also in the public area on the Unix system.

Deliverables
Preliminary report - This report has a target length of 2 side of A4 (excluding diagrams) and should definitely take no more than 3 sides of A4. Any metrics tool outputs may be included as appendix material. This report is to be handed directly to S.Webster and is due in on 20th January. 40%

Final Report - This report can be up to 5 sides of A4 in length again excluding diagrams. Any metrics tool outputs may be included as appendix material.60%

Set by: S.Webster QA: M.Machura

Blank

In

Original

Appendix K Example heuristic analysis

This appendix shows how measures and heuristics can be used to identify potential design problems and suggest changes. The example problems discussed are taken from the analysis of the VocabTutor system and are chosen to show a range problem types and approaches. This should help to clarify both the use of measures and heuristic for design assessment in general (as discussed in section 2.3.4) and also the mapping of the subjects' reports onto the case study database (section 5.3.4).

This section should be read in conjunction with VocabTutor design documents (appendix H) and the source code (appendix I)

K.1 Measurement results

The following tables (K-1, K-2 and K-3) show results from selected measures for the VocabTutor system. Shaded results are those that are referenced in the example issues discussed in sections K.2 and K.3 below.

CLASS	WMC	CBO	LCOM	RFC	DIT	NOC
ContainedHint	4	2	6	8	2	0
ContainsHint	4	2	6	8	2	0
HelpScreen	8	7	28	15	5	0
Hint	7	2	0	10	1	5
HintVector	5	5	8	15	2	0
LanguageTutorFrame	10	5	45	16	5	3
LearnScreen	4	20	0	47	6	0
RemindHint	4	2	6	8	2	0
RhymeHint	4	2	6	8	2	0
SentenceHint	4	2	6	11	2	0
TestScreen	4	15	2	29	6	0
Tutor	2	4	1	9	1	0
VocabElement	24	11	0	47	1	0
Vocabulary	7	11	0	25	1	0
WelcomeScreen	4	11	2	24	6	0

Table K-1 Chidamber and Kemerer metrics results

Class	NOCV	NOIV	NOIM	NOPIM	CC
ContainedHint	0	0	4	4	5
ContainsHint	0	0	4	4	5
HelpScreen	0	2	8	8	8
Hint	1	3	7	7	9
HintVector	5	0	5	5	12
LanguageTutorFrame	0	1	10	10	6
LearnScreen	0	21	4	3	43
RemindHint	0	0	4	4	5
RhymeHint	0	0	4	4	5
SentenceHint	0	0	4	4	10
TestScreen	0	16	4	4	28
Tutor	3	0	1	1	7
VocabElement	1	5	24	24	67
Vocabulary	0	4	7	7	32
WelcomeScreen	0	9	4	3	19

Table K-2 Selected Lorenz & Kidd class-level metrics results

Class	Method	NOMS
LearnScreen	actionPerformed(ActionEvent)	32
TestScreen	actionPerformed(ActionEvent)	13
TestScreen	TestScreen()	12
Vocabulary	loadVocabulary(String)	12
LearnScreen	LearnScreen()	11
VocabElement	VocabElement(String)	11
WelcomeScreen	WelcomeScreen()	11

Table K-3 Lorenz & Kidd method size metrics results (top-end subset)

K.2 Real problems

This section discusses a number of heuristic violations in the VocabTutor system that were considered to indicate real problems with the system.

Issue 1

VocabElement violates Riel's heuristic 'do not create god classes/objects in your system' (Riel 1996). This is shown by the values of three of the CK metrics and one of Lorenz and Kidds metrics.

- Weighted methods per class (WMC - 24): Largest class in the system
- Response for a class (RFC - 47), Class coupling (CC - 67): Highest degree of coupling with other classes

- Coupling between objects (CBO - 11): Coupling with a large number of distinct classes (3rd highest)

These results could be indicative of at least two problems:

- Poor allocation of responsibility, i.e., the VocabElement class contains functionality that belongs in other classes.
- Insufficient class decomposition, i.e., the VocabElement class should be broken-down into 2 or more classes.

Issue 2

HintVector violates the heuristic 'a well defined class provides a crisp abstraction of something drawn from the problem domain or solution domain' (Booch 1996a). Here the heuristic is semantic and the result is not supported by any metric results. The assessor's judgement on the meaning and quality of the abstraction must be used to detect this issue.

Issue 3

LearnScreen violates Lorenz & Kidd's heuristic 'a UI class should have no more than 9 instance variables' (Lorenz and Kidd 1994). This is shown by value of Lorenz & Kidd's number of instance variables metric (NOIV - 21). This indicates insufficient decomposition in the interface classes.

Issue 4

Vocabulary.loadVocabulary(String) violates Lorenz & Kidd's heuristic 'a method should not send more than 9 messages' (Lorenz and Kidd 1994). This is shown by value of Lorenz & Kidd's number of message sends metric (NOMS - 12). This indicates either function-oriented design or poor allocation of responsibilities. It can be seen from table K-3 that a total of 7 methods fall into this category.

Issue 5

The similarity between the value for *NOIM and NOPIM shows that* there is little use of protected or private methods (2 only). This may indicate a lack decomposition of

methods within the class (the NOMS values from table K-3 also support this view). Alternatively, there may be methods in the public interface of classes that should not have been declared public.

K.3 False problems

As has previously been discussed, a heuristic violation does not necessarily indicate a real problem with the system under investigation. This section lists example heuristic violations found in the VocabTutor system that were considered not to indicate real problems.

Non-issue 1

HintVector violates Lorenz & Kidd's heuristic 'a class should have no more than 3 class variables' (Lorenz and Kidd 1994). This is shown by value of Lorenz & Kidd's number of class variables metric (NOCV - 5). However examination of the HintVector class showed that these 5 class variables were in fact constants (denoted 'final' in Java). The cause of this issue could be either an oversight in the definition of the metric counting rules, or applying the heuristic out of context. In either case, this was considered as a 'non-issue'.

Non-issue 2

LanguageTutorFrame violates the heuristic 'any one module should represent a single, coherent concept, at a uniform level of abstraction' (Berard 1993). A number of the case studies subjects used the LCOM metric as a justification for this heuristic violation (LCOM - 45). However, an examination of the semantics of the class didn't show any cohesion problems. It was felt that LCOM didn't really capture the level cohesion of the class and therefore this was considered as a 'non-issue'.

Appendix L Data model for the case study database

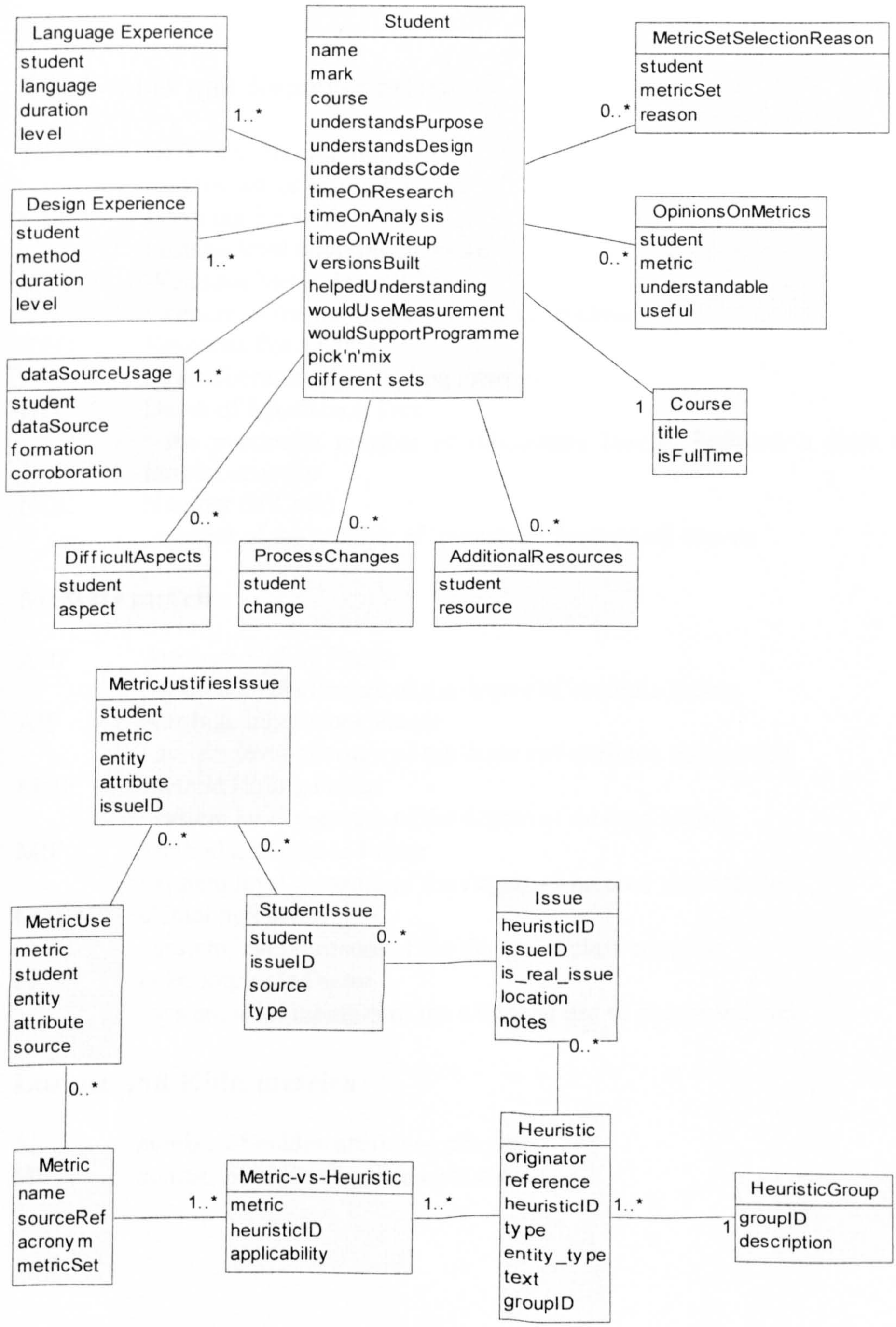


Figure L-1 Data model for case study database

Blank

In

Original

Appendix M Description of metrics used

Chidamber and Kemerer metrics

LCOM Lack of COhesion in Methods
: intra-class cohesion measure

CBO Coupling Between *Objects*
: a class level coupling measure

WMC Weighted Methods per Class
: a count of the number of methods in a class

RFC Response For a Class
: a size/complexity/coupling measure

DIT Depth of Inheritance Tree
: the maximum number of *inheritance* levels between a class to it's farthest ancestor

NOC Number Of Child
: a count of the number of immediate descendant classes

MOOD metrics

AHF Attribute Hiding Factor
: system level measure of the degree of attribute hiding

AIF Attribute inheritance Factor
: system level measure of the degree of attribute inheritance

MHF Method Hiding Factor
: system level measure of the degree of method hiding

MIF Method Inheritance Factor
: system level measure of the degree of method inheritance

COF COupling Factor
: system level measure of the degree of class coupling

PF Polymorphism Factor
: system level measure of the extent of use of polymorphism

Lorenz and Kidd metrics

Ah number of hidden attributes per class

Mh number of hidden methods per class

Blank

In

Original

References

Abreu, F. B. 'Metrics for object-oriented environments', in *Proc. 3rd International Conf. on Software Quality.* Lake Tahoe, Nevada: 1993.

Abreu, F. B. and R. Carapuca, 'Candidate Metrics for Object-Oriented Software within a Taxonomy Framework', *Journal of Systems & Software*, 26, pp87-96, 1994.

Abreu, F. B., M. Goulao, et al. 'Toward design quality evaluation of object-oriented software systems', in *Proc. 5th International conference on software quality.* Austin, Texas, USA: 1995.

Abreu, F. B. and W. Melo. 'Evaluating the impact of object-oriented design on software quality', in *Proc. 3rd International software metrics symposium.* Berlin, Germany: IEEE, 1996.

Albrecht, A. J. 'Measuring Application Development Productivity', in *Proc. SHARE-GUIDE Symposium.* Monterey, CA: IBM, 1979.

Banker, R. D., R. J. Kauffman, et al., 'Repository evaluation of software reuse', *IEEE Transactions on Software Engineering*, 19(4), pp379-389, 1993.

Bar-David, T., 'Practical consequences of formal definitions of inheritance', *Journal of Object Oriented Programming*, (July/August), pp43-49, 1992.

Basili, V., G. Caldiera, et al., *The goal question metric approach,* in *Encyclopedia of software engineering*, Wiley: 1994.

Basili, V. R., L. Briand, et al., A validation of object-oriented design metrics as quality indicators. No. CS-TR-3443, University of Maryland, 1995.

Basili, V. R., L. C. Briand, et al., 'A validation of object-oriented design metrics as quality indicators', *IEEE Transactions on Software Engineering*, 22(10), pp751-761, 1996.

Basili, V. R. and H. D. Rombach, 'The TAME project: Towards Improvement-oriented software environments', *IEEE Transactions on Software Engineering*, 14(6), pp758-771, 1988.

Belkhouche, B. and M. Chavarro, 'Analysis of object-oriented designs', *Journal of Object Oriented Programming*, 7(9), pp30-42, 1995.

Berard, E. V., *Essays on object-oriented software engineering*. Prentice Hall: 1993.

Berdie, R. D., J. F. Anderson, et al., *Questionnaires: design and use*. 2 ed. The Scarecrow Press, Inc.: 1996.

Bieman, J. M. and L. M. Ott, 'Measuring functional cohesion', *IEEE Transactions on Software Engineering*, 20(8), pp644-657, 1994.

Bieman, J. M. and J. X. Zhao, 'Reuse through inheritance: a quantitative study of C++ software', *Proc. ACM Software Reusability Symp. (SRS'94)*, (April), 1995.

Blaha, M., W. Premerlani, et al., 'Converting OO models into RDBMS schema', *IEEE Software*, 11(3), pp28-39, 1994.

Boehm, B. W., J. R. Brown, et al., *Characterisics of software quality*. TRW Series of Software Quality: Amsterdam, North Holland, 1978.

Booch, G., *Object-Oriented Design: with applications*. Benjamin Cummings: 1991.

Booch, G., *Object-Oriented Design: with applications*. 2nd ed. Benjamin Cummings: 1994.

Booch, G., 'The unified modeling language', *Unix review*, 14(13), pp41-4,46,48, 1996.

Briand, L., E. Arisholm, et al., Empirical studies of object-oriented artifacts, methods and processes: State of the art and future directions. Technical report No. ISERN-99-12, ISERN, 1999.

Briand, L., K. El Emam, et al., 'On the application of measurement theory in software engineering', *Empirical Software Engineering: an international journal*, 1(1), 1996.

Briand, L., T. Langley, et al., A replicated assessment and comparison of common software cost modeling techniques. Technical report No. ISERN-99-15, Fraunhofer Institute for Experimental Software Engineering, 1999.

Briand, L. C., C. Bunse, et al., 'A contolled experiment for evaluating guidelines on the maintainability of object-oriented designs', *IEEE Transactions on Software Engineering*, 27(6), pp513-530, 2001.

Briand, L. C., J. Daly, et al. 'Predicting fault-prone classes with design measures in object-oriented systems', in *Proc. Proceedings Ninth International Symposium on Software Reliability Engineering*. IEEE Comput. Soc, 1998a.

Briand, L. C., J. W. Daly, et al., 'A unified framework for cohesion measurement in object-oriented systems', *Empirical Software Engineering*, 3(1), pp65-117, 1998b.

Briand, L. C., J. W. Daly, et al., 'A unified framework for coupling measurement in object-oriented systems', *IEEE Transactions on Software Engineering*, 25(1), pp91-121, 1999.

Brown, W. J., R. C. Malveau, et al., *AntiPatterns: refactoring software, architectures and projects in crisis*. Wiley: 1998.

Budgen, D., *Software design*. Addison-Wesley: 1994.

Bunge, M., *Treatise on basic philosophy: ontology I: The furniture of the world.* Riedel: Boston, 1977.

Bunge, M., *Treatise on basic philosophy: ontology II: The world of systems*. Riedel: Boston, 1979.

Cain, A., *JMetric*. (Web Site) Swinburne University of Technology. URL: http://jmetric.it.swin.edu.au/products/jmetric/. [Accessed 01/06/99]

Cartwright, M. and M. Shepperd. 'Building predictive models from object-oriented metrics', in *Proc. 8th European Software Control and Metrics Conf.* Berlin: 1997.

Cartwright, M. H., An empirical investigation into metrics for object-oriented software. Ph.D. Thesis, Computing Department, Bournemouth University, 1998.

Chalmers, A. F., *What is this thing called science*. 2nd ed. Open University Press: 1982.

Chidamber, S. R., D. P. Darcy, et al., 'Managerial use of object oriented software metrics: an exploratory analysis', *IEEE Transactions on Software Engineering*, 24(8), pp629-639, 1998.

Chidamber, S. R. and C. F. Kemerer, 'Towards a metrics suite for object oriented design', *SIGPLAN Notices*, 26(11), pp197-211, 1991.

Chidamber, S. R. and C. F. Kemerer, 'A metrics suite for object oriented design', *IEEE Transactions on Software Engineering*, 20(6), pp476-493, 1994.

Chidamber, S. R. and C. F. Kemerer, 'Authors' reply', *IEEE Transactions on Software Engineering*, 21(4), pp265, 1995.

Churcher, N. and M. J. Shepperd, 'Comment on "A metrics suite for object oriented design"', *IEEE Transactions on Software Engineering*, 21(3), pp263-265, 1995a.

Churcher, N. I. and M. J. Shepperd, 'Towards a Conceptual Framework for Object Oriented Software Metrics', *ACM SIGSOFT Software Engineering Notes*, 20(2), pp69-76, 1995b.

Civello, F. 'Roles for composite objects in object-oriented analysis and design', in *Proc. OOPSLA 93*. Addison-Wesley, 1993.

Coad, P., 'OOD Criteria, Part1', *Journal of Object Oriented Programming*, (June), pp67-70, 1991a.

Coad, P., 'OOD Criteria, Part2', *Journal of Object Oriented Programming*, (July/August), 1991b.

Coad, P., 'OOD Criteria, Part3', *Journal of Object Oriented Programming*, (September), pp67-70, 1991c.

Coad, P. and E. Yourdon, *Object-Oriented Analysis*. Second Edition ed. Yourdon Press: 1991a.

Coad, P. and E. Yourdon, *Object-Oriented Design*. Yourdon Press/Prentice Hall: 1991b.

Cook, S. and J. Daniels, *Designing object systems : object-oriented modelling with Syntropy*. The object-oriented series, Prentice Hall: 1994.

Cook, W. R. 'Interfaces and Specifications for the Smalltalk-80 Collection Classes', in *Proc. OOPSLA '92*. Vancouver, Canada: 1992.

Coplien, J. O., *Advanced C++ : programming styles and idioms*. Adison Wesley: 1992.

Coppick, J. C. and T. J. Cheatham. 'Software metrics for object-oriented systems', in *Proc. Annu. ACM Computer Science Conf.* Kansas: ACM Press, 1992.

Creswell, J. W., *Qualitative inquiry and research design: choosing among five traditions*. Sage Publications: 1998.

Curtis, B., M. I. Kellner, et al., 'Process modeling', *Communications of the ACM*, 35(9), pp75-90, 1992.

Daly, J., A. Brooks, et al. 'The effect of inheritance on the maintainability of object-oriented software: an empirical study', in *Proc. Int. Conf. on Softw. Maintenance ICSM'95*. Nice: 1995.

Daly, J., A. Brooks, et al. 'Evaluating the effect of inheritance on the maintainability of object-oriented software', in *Proc. Empirical Studies of Programmers*. Washington, DC: 1996.

de Champeaux, D., *Object-oriented development process and metrics*. Prentice Hall International: 1997.

DeMarco, T., *Controlling software projects*. Yourdon Press: New York, 1982.

Dvorak, J., 'Conceptual Entropy and Its Effect on Class Hierarchies.', *IEEE Computer*, (June), pp59-63, 1994.

El Emam, K., 'Benchmarking Kappa: interrater agreement in software process assessments', *Empirical Software Engineering*, 4(2), pp113-33, 1999.

El Emam, K. 'The reliability and validity of software process assessments', in *Proc. Empirical Assessment in Software Engineering.* Keele, UK: 2001.

Fenton, N. E., *software metrics: a rigorous approach.* Chapman and Hall: London, UK, 1991.

Fenton, N. E., S. Lawrence Pfleeger, et al., 'Science and substance: a challenge to software engineers', *IEEE Software*, (July), pp86-95, 1994.

Fenton, N. E. and S. L. Pfleeger, *Software metrics: a rigorous and practical approach.* 2nd ed. International Thompson Computer Press: London, UK, 1996.

Fink, A. and J. Kosecoff, *How to conduct surveys.* Sage publications: 1985.

Finkelstein, L. and M. S. Leaning, 'A review of the fundamental concepts of measurement', *Measurement*, 2(1), pp25-34, 1984.

Firesmith, D., 'An Expanded View of Messages', *Journal of Object Oriented Programming*, (July/August), pp51-52, 1993.

Firesmith, D., 'Inheritance Guidelines', *Journal of Object-Oriented Programming*, (May), pp67-72, 1995.

Fowler, F. J., *Improving survey questions: design and evaluation.* Applied social research methods series, Sage Publications: 1995.

Gamma, E., R. Helm, et al., *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison -Wesley: 1994.

Gibbon, C., Heuristics for object-oriented design. Ph.D. Thesis, Computer Science Department, University of Nottingham, 1997.

Gibbon, C. and C. Higgins, 'Teaching object-oriented design with heuristics', *SIGPLAN Notices*, 31(7), pp12-16, 1996.

Gilb, T., *Tools for design by objectives,* in *Software requirements specification and testing,* T. Anderson, Editor, Blackwell Scientific: 1986.

Gilb, T., *Principles of software engineering management.* Addison Wesely: Reading, Mass., USA, 1988.

Grady, R., *Practical software metrics for project management and process improvement.* Prentice-Hall: 1992.

Grady, R. B., 'Measuring and managing the software maintenance process', *IEEE Software*, 4(5), pp35-45, 1987.

Grady, R. B., 'Successfully applying software metrics', *IEEE Computer*, 27(9), pp18-25, 1994.

Grady, R. B. and D. L. Caswell, *Software metrics: Establishing a company-wide program.* Prentice Hall: Englewood Cliffs, NJ, 1987.

Graham, I., *Migrating to Object Technology.* Addison-Wesley: 1995a.

Graham, I. 'Progress with object-oriented metrics', in *Proc. Object Expo Europe.* London: 1995b.

Graham, I., 'Making progress in metrics', *Object magazine*, 6(8), pp68-73, 1996.

Greenwood, D. J. and L. Morten, *Introduction to action research.* Sage Publications: 1998.

Hall, T., 'Software quality programmes: a snapshot of theory versus reality', *Software quality journal*, 5(4), pp235-242, 1996.

Hall, T. and N. Fenton, 'Implementing effective software metrics programs', *IEEE Software*, 14(2), pp55-65, 1997.

Hall, T. and N. E. Fenton, 'Implementing software metrics - the critical success factors', *Software Quality Journal*, 3, pp195-208, 1994.

Halstead, M. H., *Elements of Computer Science*. Elsevier North-Holland: 1977.

Harrison, R., S. J. Councell, et al., 'An evaluation of the MOOD set of object-oriented software metrics', *IEEE transactions on software engineering*, 24(6), pp491-496, 1998.

Harrison, R. and R. Nithi. 'An empirical evaluation of object-oriented design metrics', in *Proc. OOPSLA'96 - workshop: OO product metrics*. 1996.

Haythorn, W., 'What is Object-Oriented Design', *Journal of Object Oriented Programming*, (March/April), pp67-78, 1994.

Henderson-Sellers, B., 'OO metrics programme', *Object Magazine*, (October), pp73-79,95, 1995.

Henderson-Sellers, B., *Object-oriented metrics: measures of complexity*. Prentice Hall: 1996a.

Henderson-Sellers, B., *OPEN and the OPEN/MeNtOR project*. (Web Page) Swinburne University. URL: http://www.csse.swin.edu.au/cotar/OPEN/open1.html. [Accessed 8/11/97]

Henderson-Sellers, B., L. Constantine, et al., 'Coupling and cohesion (towards a valid metrics suite for object-oreinted analysis and design)', *Object-oriented systems*, 3(3), pp143-158, 1996.

Henderson-Sellers, B. and I. Graham, 'OPEN: toward method convergence', *Computer*, 29(4), pp86-89, 1996.

Henderson-Sellers, B., I. M. Graham, et al., 'Methods unification: the open methodology', *JOOP*, 10(2), pp41-43, 55, 1997.

Henry, S. and D. Kafura, 'Software quality metrics based on inter-connectivity', *Journal of Systems & Software*, 2(2), pp121-131, 1981a.

Henry, S. and D. Kafura, 'Software structure metrics based on information flow', *IEEE Transactions on Software Engineering*, 7(5), pp510-517, 1981b.

Hitz, M. 'Measuring reuse attributes in object-oriented systems', in *Proc. OOIS'95*. Dublin City University: Springer-Verlag, 1995.

Hitz, M. and B. Montazeri. 'Measuring coupling and cohesion in object-oriented systems', in *Proc. Int. Symposium on Applied Corporate Computing*. Monterrey, Mexico: 1995a.

Hitz, M. and B. Montazeri. 'Measuring product attributes of object-oriented systems', in *Proc. ESEC `95 (5th European Software Engineering Conference). Barcelona, Spain*: Springer Verlag 1995, 1995b.

Hitz, M. and B. Montazeri, 'Chidamber & Kemerer's metrics suite: A measurement theory perspective', *IEEE Transactions on Software Engineering*, 22(4), pp267-71, 1996.

Hopkins, T., *Complexity metrics for quality assessment of object-oriented design,* in *Software Quality Management II, vol. 2: Building Quality into Software,* P. M. Ross, C. A. Brebbia, G. Staples and J. Stapleton, Editor, Computational Mechanics Press: 1994.

Hopkins, T. P., 'Do we need object-oriented design metrics?', *Hotline on Object-Oriented Technology,* 2(8), pp16-17, 1991.

Hursch, W. L. 'Should Superclasses be Abstract?', in *Proc. European Conference on Object_Oriented Programming.* Springer Verlag, 1994.

IFPUG, Function Point Counting Practices Manual: Release 3.4. No. , International Function Point User's Group, 1992.

ISO9001, Quality systems - Assurance model for design/development, production installation and servicing capability. Standard No. 9001 (Part 1), International Organization for Standardization, 1987.

Jeffery, R. and M. Berry. 'A framework for evaluation and prediction of metrics programme success', in *Proc. 1st Intl Metrics Symposium.* Baltimore, U.S.A.: IEEE Computer Press, 1993.

Keyes, J., 'New metrics needed for new generation', *Software Magazine,* 12(6), pp42-56, 1992.

Kirsopp, C., *Automated Support for Assessing OO Architectures.* (Presentation slides) Bournemouth University.
URL:http://dec.bournemouth.ac.uk/~ckirsopp/OOPS_116.htm. [Accessed 07/02/00]

Kirsopp, C. 'Measurement and the software development process', in *Proc. 12th European Software Control and Metrics Conference.* London, UK: 2001a.

Kirsopp, C., A survey and review of object-oriented design heuristics. Tech Report No. ESERG TR-01-04, Bournemouth University, 2001b.

Kirsopp, C. and S. Webster. 'Object-Oriented Design Heuristics: BOF Session', in *Proc. Object Technology '97*. Oxford, UK: 1997.

Kitchenham, B., S. L. Pfleeger, et al., 'Towards a framework for software measurement validation', *IEEE transactions on software engineering*, 21(12), pp929-944, 1995.

Kitchenham, B. A., 'Towards a constructive quality model Part I : Software quality modelling, measurement and prediction', *Software Engineering Journal*, 2(4), pp105-113, 1987.

Kogure, M. and Y. Akao, 'Quality function deployment and CWQC in Japan', *Quality Progress*, (October), pp25-29, 1983.

Korson, T. and J. D. McGregor, 'Technical criteria for the specification and evaluation of object-oriented libraries', *Software Engineering Journal*, (March), pp85-94, 1992.

Krantz, D. H., R. D. Luce, et al., *Foundations of Measurement*. Vol. 1. Academic Press: 1971.

Krantz, D. H., R. D. Luce, et al., *Foundations of Measurement*. Vol. 2. Academic Press: 1989.

Krantz, D. H., R. D. Luce, et al., *Foundations of Measurement*. Vol. 3. Academic Press: London, 1990.

Kyburg, H. E., *Theory and Measurement*. Cambridge Univ. Press: Cambridge, England, 1984.

Land, L., R. Jeffery, et al., Validating the defect detection performance advantage of group designs for software reviews: report of a replicated experiment. Technical report No. ISERN-97-16, Centre for Advanced Empirical Software Research, 1997.

Leavens, G. T., 'Modular Specification and Verification of Object-Oriented Programs', *IEEE Software*, (July), pp72-80, 1991.

Lee, H. J. and W. T. Tsai, 'A new Partial Inheritance Mechanism and its Applications', *Journal of Object Oriented Programming*, (July/August), pp53-63, 1993.

Lee, Y. S., B. S. Liang, et al. 'Some Complexity Metrics for Object-Oriented Programs Based on Information Flow', in *Proc. Proceedings, Computers in Design, Manufacturing and Production.* Pris-Evry, France: IEEE Computer Society Press, 1993.

Li, W., Applying Software Maintenance Metrics in the Object-Oriented Software Development Life Cycle. Ph.D. Thesis, Virginia Polytechnic Inst. and State Univ., 1992.

Li, W. and S. Henry. 'Maintenance metrics for the object oriented paradigm', in *Proc. 1st Intl. Software Metrics Symposium.* Baltimore: IEEE Computer Society, 1993a.

Li, W. and S. Henry, 'Object-oriented metrics that predict maintainability', *Journal of Systems and Software*, 23(2), pp111-122, 1993b.

Li, W. and S. Henry, 'An empirical study of maintenance activities in two object-oriented systems', *Journal of Software Maintenance Research and Practice*, 2(2), pp131-147, 1995.

Li, W., S. Henry, et al., 'Measuring object-oriented design', *JOOP*, 8(4), pp48-55, 1995.

Liu, C.-P., B.-S. Liang, et al. 'A validation of software complexity metrics for object-oriented programs', in *Proc. International Computer Symposium 1994, (ICS'94).* Taiwan, R.O.C.: 1994.

Lorenz, M., *Object Oriented Software Development: A Practical Approach.* ed. B. Meyer. Prentice Hall: 1993.

Lorenz, M. and J. Kidd. 'OO metrics position paper', in *Proc. OOPSLA `93 Workshop on Processes and Metrics for Object Oriented Software Development.* Washington DC: 1993.

Lorenz, M. and J. Kidd, *Object-Oriented Software Metrics.* Object-Oriented Series, Prentice Hall: Englewood Cliffs, N.J., 1994.

Love, T., 'Timeless Design of Information Systems', *Object Magazine,* (November/December), pp42-48, 1991.

Martin, R., *Designing Object Oriented C++ Applications Using the Booch Method.* Vol. ISBN 0-13-203837-4. Prentice Hall: Englewood Cliffs, New Jersey, 1995.

McCabe, T. J., 'A software complexity measure', *IEEE transactions on software engineering,* 2(4), pp308-20, 1976.

McCall, J. A., P. K. Richards, et al., Factors in software quality. No. NTIS AD/A-049 014, NTIS AD/A-049 015, NTIS AD/A-049 055, US Roma Air Development Center, 1977.

McGregor, J. D. and T. Korson, 'Supporting dimensions of classification in object-oriented design', *Journal of Object Oriented Programming,* (February), pp25-30, 1993.

Melo, W. L., L. C. Briand, et al., Measuring the impact of reuse on quality and productivity in objectt-oriented systems. Technical Report No. CS-TR-3395, University of Maryland, Dept of Computer Science, 1995.

Miller, G., M. Woof, et al., Project MOOD: Business structure. No. PS-MOOD-CS-000003, Parallax Solutions Limited, 1996a.

Miller, G., M. Woof, et al., Project MOOD: Requirements capturing approach. No. PS-MOOD-CS-000002, Parallax Solutions Limited, 1996b.

Miller, J. 'Can results from software engineering experiments be safely combined', in *Proc. Sixth international software metrics symposium.* Boca Raton, Florida: IEEE Computer Society, 1999.

Myers, G. J., *Composite/Structured Design.* Van Nostrand Reinhold: New York, 1978.

Nehaniv, C. L., ed. *Computation for metaphors, analogy, and agents.* Springer-Verlag: Berlin, Germany, 1999.

Number-Six-Software, *MetricsOne.* (Measurement tool) Number Six Software, Inc. URL: http://www.numbersix.com/metricsone/. [Accessed 07/02/00]

Parnas, D. L., P. C. Clements, et al. 'Enhancing reusability with information hiding', in *Proc. ITT Workshop on reusability in programming.* 1983.

Paulk, M. C., B. Curtis, et al., 'Capability maturity model, version 1.1', *IEEE Software,* 10(July), pp18-27, 1993.

Payne, S. L., *The art of asking questions.* Princeton University Press: Princeton, New Jersey, 1951.

Pfleeger, S. L., 'Lessons learned in building a corporate metrics program', *IEEE Software*, 10(5), pp67-74, 1993.

Pfleeger, S. L., 'Experimental design and analysis in software engineering', *Software Engineering Notes*, 19-20(vol.19 no.4 to vol.20 no.5), 1994.

Pfleeger, S. L., 'Maturity, models and goals: how to build a metrics plan', *Journal of systems and software*, 31(2), pp143-155, 1995.

Pickard, L. M., B. A. Kitchenham, et al., 'Combining empirical results in software engineering', *Information and Software Technology*, 40(14), 1998.

Ponder, C. and B. Bush, 'Polymorphism considered harmful', *ACM SIGSOFT Software Engineering Notes*, 19(3), pp35-37, 1994.

Pressman, R. S. and D. Ince, *Software engineering: a practitioners approach*. 5th ed. McGraw-Hill: 2000.

Rajaraman, C. and M. R. Lyu. 'Reliability and maintainability related software coupling metrics in C++ programs', in *Proc. 3rd Intl. Symposium on Software Reliability Engineering*. IEEE Computer Press, 1992a.

Rajaraman, C. and M. R. Lyu. 'Some coupling measures for C++ programs', in *Proc. 8th Intl. Conf. Technology of Object-Oriented Languages and Systems*. Santa Barbara: Prentice Hall, 1992b.

Rational, *UML Resource Center*. (Web page) Rational Software Corporation. URL: http://www.rational.com/uml/index.jtmpl. [Accessed 07/02/00]

Reyes, L. and D. Carver. 'Predicting object reuse using metrics', in *Proc. SEKE '98. Tenth International Conference on Software Engineering and Knowledge Engineering*. Knowledge Syst. Inst, Skokie, IL, USA: 1998.

Riel, A. J. 'Tutorial 38 handout : object-oriented design though heuristics', in *Proc. OOPSLA '94*. Portland, Oregon: 1994.

Riel, A. J., *Object oriented design heuristics*. Addison Wesley: Reading, Massachusets, USA, 1996.

Rising, L., OO Design Heuristics. Internal Company Document No. , AG Communication Systems, 1994.

Roberts, F. S., *Measurement theory with applications to decision making, utility, and social sciences*. Adison-Wesley: Reading , MA, 1979.

Rombach, H. D. and V. R. Basili. 'Practical benefits of goal-oriented measurement', in *Proc. Annual Workshop of the Centre for Software Reliability: Reliability and Measurement*. Garmisch-Partenkirchen, Germany: Elsevier, 1990.

Rosson, M. B. and E. Gold, 'Problem-solution mapping in object-oriented design', *SIGPLAN notices*, 24(10), pp7-10, 1989.

Rumbaugh, J., 'Disinherited! Examples of Misuse of Inheritance', *Journal of Object Oriented Programming*, (Febuary), pp22-24, 1993.

Rumbaugh, J., 'A search for values : attributes and associations', *Journal of object oriented programming*, (June), pp6-8, 49, 1996.

Sharble, R. C. and S. S. Cohen, 'The Object-Oriented Brewery: A Comparison of Two Object_Oriented Development Methods', *SIGSOFT Software Engineering Notes*, 18(2), pp60-73, 1993.

Shepperd, M., *Foundations of software measurement*. Prentice Hall: 1995.

Shepperd, M. and D. Ince. 'Algebraic validation of software metrics', in *Proc. 3rd european software engineering conference*. Milan: Springer-Verlag, 1991.

Shepperd, M. J. and D. C. Ince, *Derivation and validation of software metrics*. Open University Press: 1993.

Siegel, S. and N. Castellan, *Nonparametric statistics for the behavioural sciences*. 2 ed. McGraw-Hill: New York, 1988.

Sommerville, I., *Software engineering*. 6th ed. Addison-Wesley: 2001.

Tegarden, D. P., S. D. Sheetz, et al. 'Effectiveness of Traditional Software Metrics for Object-Oriented Systems', in *Proc. Twenty-Fifth Hawaii International Conference on System Sciences*. Hawaii: IEEE, 1992.

Tegarden, D. P., S. D. Sheetz, et al., 'A software complexity model of object-oriented systems', *Decision Support Systems: the International Journal*, , 1993.

Velleman, P. F. and L. Wilkinson, 'Nominal, ordinal, interval and ratio typologies are misleading', *The American Statistician*, 47(1), pp65-72, 1993.

Visaggio, G., Assessing the maintenance process through replicated, controlled experiment. technical report No. ISERN-97-27, University of Bari, 1997.

von Mayrhauser, A. and A. M. Vans, 'Industrial experience with an integrated code comprehension model', *Software engineering journal*, 10(5), pp171-182, 1995.

Webster, S., *On the evolution of OO methods*. 1996, Bournemouth University: Unpublished.

Wei, L., Applying Software Maintaining Metrics in the Software Development Life Cycle. Masters Thesis, MIT, 1992.

Weyuker, E. J., 'Evaluating Software Complexity Measures', *IEEE Transactions on Software Engineering*, 14(9), pp1357-1365, 1988.

Whitmire, S. A. 'A formal object model for measurement', in *Proc. OOPSLA'96 - workshop: OO product metrics*. 1996.

Whitmire, S. A., *Object-oriented design measurement*. John WIley & Sons, Inc.: 1997.

Wilkie, F. G. and B. Hylands, 'Measuring complexity in C++ software', *Software practice and experience*, 28(5), pp513-546, 1998.

Wirfs-Brock, A. and B. Wilkerson, 'Variables Limit Reusability', *Journal of Object-Oriented Programming*, 2(1), pp34-40, 1989.

Wohlin, C. 'Software inspection benchmarking: a feasability study and illustration of opportunities', in *Proc. Empirical assessment in software engineering (EASE)*. Keele, UK: 2001.

Wohlin, C., P. Runeson, et al., *Experimentation in software engineering: an introduction*. The Kluwer internation series in software engineering, ed. V. R. Basili. Kluwer Academic Publishers: 2000.

Yin, R. K., *Applications of case study research*. Applied social research methods series, Sage Publications: 1993.

Yin, R. K., *Case study research: design and methods*. 2nd ed. Applied social research methods series, Sage Publications: 1994.

Yourdon, E., *Object-Oriented Systems Design: An Integrated Approach*. Prentice Hall International Editions: 1994.

Zhao, M., C. Wohlin, et al., 'A comparison between software design and code metrics for the prediction of software fault content', *Information and Software Technology*, 40(14), pp801-9, 1998.

Zuse, H., *Software Complexity. Measures and Methods*. de Gruyter: Berlin, 1991.

Zuse, H., *Foundations of the validation of object-oriented software measures,* in *Theorie und Praxis der Softwaremessung,* H. Zuse and D. Reiner, Editor, DUV Deutsche Universitätsverlag: 1994.

Zuse, H., *A framework of software measurement.* Walter de Gruyter & Co.: Berlin, 1997.

Zuse, H. and P. Bollmann, 'Software metrics: using measurement theory to describe the properties and scales of static complexity metrics', *ACM SIGPLAN Notices*, 24(8), pp23-33, 1989.