Class and Object Modularity
Description and Measurement

Christine McClean

Doctor of Philosophy in Engineering

2006

CERTIFICATE OF AUTHORSHIP/ORIGINALITY

I certify that the work in this thesis has not previously been submitted for a degree nor has it been submitted as part of requirements for a degree except as fully acknowledged within the text.

I also certify that the thesis has been written by me. Any help that I have received in my research work and the preparation of the thesis itself has been acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

Signature of Candidate

_____

Acknowledgements

Thank you to my supervisors John Leaney and Keiko Yasukawa for their support, valuable suggestions and comments regarding this research. In particular, I would like to express my very special thanks and gratitude to Keiko Yasukawa for her help and support throughout the process of revising this thesis. “I couldn’t have done it without you!”

Many thanks also to Keiko Yasukawa, Noel McClean and John Leaney for their assistance in proof reading this thesis document and suggesting improvements.

Thank you to the anonymous assessors of this thesis for their helpful comments and insights that have contributed to many improvements to this research work.

Thank you to Peter Petocz for suggesting the calculation of an aggregation of modularity as an appropriate data analysis technique.

Thank you to Adrian Richards for offering insights into the process of scientific measurement.

Thank you to Scientific Toolworks Inc. for providing a copy of the Understand For C++ code analysis application.

Finally, last but not least, thank you to my husband Noel and our girls Catie and Merryn for their support, patience and understanding throughout this research.

Table of Contents

1. INTRODUCTION 1

1.1. APPROACH TAKEN TO SYSTEMATIC MEASURE DEVELOPMENT PROCESS DEFINITION 2
1.2. KEY CONCEPTS 3
1.3. MEASURE DEVELOPMENT PROCESS 6
1.4. MEASURE VALIDITY 11
1.4.1 Content validation 14
1.4.2 Construct validation 16
1.5. MEASURE RELIABILITY 18
1.6. LEVEL OF MEASUREMENT 18
1.7. COMMENTS 19
1.8. STRUCTURE OF THIS THESIS 20

2. LITERATURE REVIEW 22

2.1. PROCESSES OF SOFTWARE MEASURE DEVELOPMENT 23
2.2. PROCESSES CURRENTLY USED TO DEVELOP DESCRIPTIVE SOFTWARE MEASURES 27
2.3. THEORETICAL BASIS OF DESCRIPTIVE SOFTWARE MEASURE DEVELOPMENT 33
2.4. CONCLUSION 35

3. CONCEPTUAL DEFINITION 37

3.1. STAGE 1 OF MEASURE DEVELOPMENT PROCESS - CONCEPTUAL DEFINITION 37
3.1.1 Prerequisites to the conceptual definition stage 40
3.1.2 Performance of the conceptual definition stage 41
3.1.3 Products of the conceptual definition stage 42
3.1.4 Practical Considerations 42
3.2. CONCEPTUAL DEFINITION OF C++ CLASS AND OBJECT MODULARITY 44
3.2.1 Prerequisites 44
3.2.2 Sub-characterisation of object-oriented software modularity 44
3.2.2.1 Direct mapping 46
3.2.2.2 Information Hiding 47
3.2.2.3 Few Interfaces 50
3.2.2.4 Explicit Interfaces 52
3.2.2.5 Small Interfaces 54
3.2.3 Conceptual definition of modularity sub-characteristics 59
3.2.3.1 Interface dependence sub-characteristic 60
3.2.3.2 External relationships sub-characteristic 62
3.2.3.3 Connection obscurity sub-characteristic 64
3.2.3.4 Dependency sub-characteristic 67
3.2.4 Validation of the sub-characterisation 69
3.3. CONCLUSION 69

4. ENTITY MODELLING 71

4.1. STAGE 2 OF MEASURE DEVELOPMENT PROCESS - ENTITY MODELLING 71
4.1.1 Prerequisites to the entity modelling stage 74
4.1.2 Performance of the entity modelling stage 74
4.1.3 Products of the entity modelling stage 75
4.1.4 Assessing mathematical model validity 76
4.1.5 Practical Considerations 76
4.2. ENTITY MODELLING OF C++ CLASS AND OBJECT MODULARITY 78
4.2.1 Prerequisites 79
4.2.2 Selection of the mathematical model 79
4.2.3 C++ class modularity 81
4.2.3.1 C++ class module model 81
4.2.3.2 Interface dependence sub-characteristic of C++ class modularity 84
4.2.3.3 External relationships sub-characteristic of C++ class modularity 86
4.2.3.4 Connection obscurity sub-characteristic of C++ class modularity 88
4.2.3.5 Dependency sub-characteristic of C++ class modularity 92
4.2.3.6 Class modularity sub-characteristics sharing identical feature descriptions 95

4.2.4 C++ object modularity 96
4.2.4.1 C++ object module model 96
4.2.4.2 Interface dependence sub-characteristic of C++ object modularity 102
4.2.4.3 External relationships sub-characteristic of C++ object modularity 106
4.2.4.4 Connection obscurity sub-characteristic of C++ object modularity 109
4.2.4.5 Dependency sub-characteristic of C++ object modularity 116
4.2.4.6 Summary: object mathematical model completeness 119
4.2.4.7 Object modularity sub-characteristics sharing identical feature descriptions 121
4.3. CONCLUSION 122

5. OPERATIONAL MEASURE DEFINITION 124

5.1. STAGE 3 OF MEASURE DEVELOPMENT PROCESS - OPERATIONAL DEFINITION 124
5.1.1 Prerequisites to operational definition stage 126
5.1.2 Performance of operational definition stage 126
5.1.3 Product of operational definition stage 126
5.1.4 Assessing measure level of measurement and validity 128
5.1.5 Practical Considerations 128
5.2. OPERATIONAL DEFINITION OF C++ CLASS AND OBJECT MODULARITY 129
5.2.1 Prerequisites 130
5.2.2 Mathematical measure definition notation 130
5.2.3 Measures of C++ class modularity 132
5.2.3.1 Measures of C++ class interface dependence 133
5.2.3.2 Measures of C++ class external relationships 139
5.2.3.3 Measures of C++ class connection obscurity 141
5.2.3.4 Measures of C++ class dependency 145
5.2.4 Measures of C++ object modularity 149
5.2.4.1 Measures of C++ object interface dependence 149
5.2.4.2 Measures of C++ object external relationships 154
5.2.4.3 Measures of C++ object connection obscurity 156
5.2.4.4 Measures of C++ object dependency 163
5.2.5 Level of measurement of modularity measures 168
5.2.6 Content validation of modularity measures 169
5.2.6.1 Example of a content validation 170
5.3. CONCLUSION 173

6. MEASUREMENT INSTRUMENT IMPLEMENTATION 175

6.1. STAGE 4 OF MEASURE DEVELOPMENT PROCESS - MEASUREMENT INSTRUMENT IMPLEMENTATION 175
6.1.1 Prerequisites to the measurement instrument implementation stage 178
6.1.2 Performance of the measurement instrument implementation stage 178
6.1.3 Products of the measurement instrument implementation stage 179
6.1.4 Assessing implemented measure level of measurement and validity 179
6.1.5 Reliability of software based measurement instruments 180
6.1.6 Practical Considerations 181
6.2. IMPLEMENTATION OF C++ CLASS AND OBJECT MODULARITY MEASURES 182
6.2.1 C++ basic software model implementation 184
6.2.1.1 Natural language C++ basic software model 184
6.2.1.2 Mathematical C++ basic software model 185
6.2.1.3 Implementation of C++ basic software model 187
6.2.2 Software measurement model implementation 189
6.2.2.1 Derivation of C++ software modularity measurement model 189
6.2.2.2 Implementation of C++ software modularity measurement model 190
6.2.3 Software modularity measures implementation 192
6.2.3.1 Class external relationships measure implementation 195
6.2.3.2 Object external relationships measure implementation 196
6.2.3.3 Object connection obscurity measure implementation 197
6.2.3.4 Object dependency measure implementation 200
6.2.4 Level of measurement and content validity of implemented measures 201
6.2.4.1 Level of measurement 201
6.2.4.2 Content validity 201
6.3. CONCLUSION 202

7. APPLICATION OF C++ CLASS AND OBJECT MODULARITY MEASURES 204

7.1. DATA REDUCTION TECHNIQUES 206
7.1.1 Calculation of an Aggregate of Modularity 206
7.1.1.1 Preliminary calculations 207
7.1.1.2 Class and object modularity aggregate calculation 209
7.1.1.3 Class and object modularity weighted aggregate calculation 213
7.1.2 Calculation of a Modularity Distance 215
7.2. CONTENT VALIDATION – EMULEPLUS SOFTWARE SYSTEM 216
7.2.1 eMulePlus object interface dependence 218
7.2.2 eMulePlus object external relationships 219
7.2.3 eMulePlus object connection obscurity 219
7.2.4 eMulePlus object dependency 222
7.3. EXAMPLE OF A CONSTRUCT VALIDATION – EMULEPLUS SOFTWARE SYSTEM 224
7.4. MODULARITY MEASUREMENT CASE STUDIES 236
7.4.1 Case study 1 - modularity of the eMulePlus software system 237
7.4.1.1 Aims 237
7.4.1.2 Application 237
7.4.1.3 Analysis and interpretation 238
7.4.1.4 Discussion 245
7.4.2 Case study 2 - interface dependence of CPartFile class and object modules 247
7.4.2.1 Aims 247
7.4.2.2 Application 247
7.4.2.3 Analysis and interpretation 248
7.4.2.4 Discussion 266
7.5. CONCLUSIONS 266

8. DISCUSSION AND SUGGESTIONS FOR FURTHER WORK 269

8.1. RESEARCH OUTCOMES 269
8.1.1 Systematic process of software descriptive measure development 270
8.1.2 Descriptive measures of C++ class and object modularity 274
8.1.3 Case study – eMulePlus software system 277
8.1.4 Summary – research contributions and considerations 278
8.1.4.1 Contributions 278
8.1.4.2 Considerations 279
8.1.5 Further work 280

1. APPENDIX 1 - ENTITY-RELATIONSHIP MODEL SET DEFINITIONS 283

1.1. ENTITIES 283
1.2. BASIC RELATIONSHIPS 285
1.3. DERIVED RELATIONSHIPS 289

2. APPENDIX 2 - BASIC MODEL TO MEASUREMENT MODEL TRANSFORMATIONS . 294

2.1. TRANSFORMATION 1 - AA AND IAA 294
2.2. TRANSFORMATION 2 - AM AND IAM 296
2.3. TRANSFORMATION 3 - *AO* AND *IAO* 296
2.4. TRANSFORMATION 4 - *IMO* 297
2.5. TRANSFORMATION 5 - *MICREADA* 297
2.6. TRANSFORMATION 6 - *MICWRITEA* 299
2.7. TRANSFORMATION 7 - *MICINVM* 299
2.8. TRANSFORMATION 8 - *MIOCREADA* 300
2.9. TRANSFORMATION 9 - *MIOCWRITEA* 300
2.10. TRANSFORMATION 10 - *MIOCINVM* 301
2.11. TRANSFORMATION 11 - *CEF* 302
2.12. TRANSFORMATION 12 - *IDA* 302
2.13. TRANSFORMATION 13 - *CIF* 303
2.14. TRANSFORMATION 14 - *FIF* 303

3. APPENDIX 3 – EMULEPLUS C++ CLASS MODULARITY TO LINES OF CODE CORRELATION DATA 304

4. APPENDIX 4 - EMULEPLUS SOFTWARE SYSTEM CONTENT VALIDATION 308

4.1. EMULEPLUS CLASS MODULARITY 308
4.2. EMULEPLUS OBJECT MODULARITY 312
4.3. EMULEPLUS CONTENT VALIDATION MEASURED VALUES 317

5. APPENDIX 5 – EMULEPLUS MODULARITY DATA VALUES 322

6. APPENDIX 6 - CPARTFILE INTERFACE DEPENDENCE DATA 334

6.1. CLASS CPARTFILE 334
6.2. OBJECT CPARTFILE 344

BIBLIOGRAPHY 356

List of Figures

FIGURE 1-1 RELATIONSHIPS BETWEEN ENTITIES, CHARACTERISTICS, FEATURES AND DESCRIPTIVE MEASURES 4
FIGURE 1-2 RELATIONSHIPS BETWEEN A SOURCE CODE ENTITY AND DESCRIPTIVE MEASURES THAT QUANTIFY LINES OF CODE TO DESCRIBE SOURCE CODE SIZE 5
FIGURE 1-3 SYSTEMATIC PROCESS OF SOFTWARE DESCRIPTIVE MEASURE DEVELOPMENT 7
FIGURE 1-4 PATHS OF DESCRIPTIVE MEASURE VALIDATION 12
FIGURE 1-5 STEPS OF DESCRIPTIVE MEASURE CONTENT VALIDATION. ADAPTED FROM SPROULL (1995, P.79). 15
FIGURE 1-6 STEPS OF DESCRIPTIVE MEASURE CONTENT VALIDATION. ADAPTED FROM SPROULL (1995, P.79). 17
FIGURE 2-1 CONTRIBUTION OF SOFTWARE DESCRIPTIVE MEASURE DEVELOPMENT PROCESS TO SOFTWARE QUALITY METRICS FRAMEWORK (IEEE COMPUTER SOCIETY 1998, P. 4). 24
FIGURE 2-2 CONTRIBUTION OF SOFTWARE DESCRIPTIVE MEASURE DEVELOPMENT PROCESS TO TAME GQM PARADIGM (BASILI 1988). 25
FIGURE 3-1 CONCEPTUAL DEFINITION STAGE OF THE MEASURE DEVELOPMENT PROCESS 37
FIGURE 3-2 PROCESS OF CONCEPTUAL DEFINITION OF CHARACTERISTIC TO BE DESCRIBED BY MEASURES 42
FIGURE 3-3 REPRESENTATION OF AN OBJECT ORIENTED MODULE WITH A HIGH LEVEL OF INFORMATION HIDING 48
FIGURE 3-4 REPRESENTATION OF OBJECT ORIENTED MODULE EXTERNAL RELATIONSHIPS 51
FIGURE 3-5 REPRESENTATION OF OBJECT ORIENTED MODULE CONNECTION OBSCURITY 53
FIGURE 3-6 REPRESENTATION OF OBJECT ORIENTED MODULE DEPENDENCY 55
FIGURE 3-7 MODULARITY SUB-CHARACTERISATION BASED ON MEYER'S (1997) RULES OF MODULARITY 57
FIGURE 3-8 INTERFACE DEPENDENCE SUB-CHARACTERISATION AND CONCEPTUAL DEFINITION 60
FIGURE 3-9 EXTERNAL RELATIONSHIPS SUB-CHARACTERISATION AND CONCEPTUAL DEFINITION 62
FIGURE 3-10 CONNECTION OBSCURITY SUB-CHARACTERISATION AND CONCEPTUAL DEFINITION 64
FIGURE 3-11 DEPENDENCY SUB-CHARACTERISATION AND CONCEPTUAL DEFINITION 67
FIGURE 4-1 ENTITY MODELLING STAGE OF THE MEASURE DEVELOPMENT PROCESS 71
FIGURE 4-2 NATURAL LANGUAGE AND MATHEMATICAL ENTITY MODELS EXPRESS THE THEORETICAL BASIS OF DESCRIPTIVE MEASURES 72
FIGURE 4-3 PROCESS OF NATURAL LANGUAGE MODELLING OF A SOFTWARE ENTITY 75
FIGURE 4-4 C++ CLASS INTERFACE AND HIDDEN ELEMENTS 83
FIGURE 4-5 C++ CLASS MATHEMATICAL MODEL 83
FIGURE 4-6 C++ CLASS INTERFACE DEPENDENCE MATHEMATICAL MODEL 86
FIGURE 4-7 C++ CLASS EXTERNAL RELATIONSHIPS MATHEMATICAL MODEL 88
FIGURE 4-8 C++ CLASS CONNECTION OBSCURITY MATHEMATICAL MODEL 91
FIGURE 4-9 C++ CLASS DEPENDENCY MATHEMATICAL MODEL 94
FIGURE 4-10 C++ OBJECT-CLASS INTERFACE AND HIDDEN ELEMENTS 99
FIGURE 4-11 C++ OBJECT-CLASS MATHEMATICAL MODEL 100
FIGURE 4-12 SOLUTION ADOPTED TO MULTIPLE ATTRIBUTES AND METHODS INHERITED FROM THE SAME CLASS 101
FIGURE 4-13 C++ OBJECT INTERFACE DEPENDENCE MATHEMATICAL MODEL 105
FIGURE 4-14 C++ OBJECT EXTERNAL RELATIONSHIPS MATHEMATICAL MODEL 108
FIGURE 4-15 C++ OBJECT CONNECTION OBSCURITY MATHEMATICAL MODEL 114
FIGURE 4-16 C++ OBJECT DEPENDENCY MATHEMATICAL MODEL 118
FIGURE 5-1 OPERATIONAL DEFINITION STAGE OF THE MEASURE DEVELOPMENT PROCESS 124
FIGURE 5-2 CHARACTERISTIC TO MEASURE RELATIONSHIP (CHARMER) DIAGRAM 127
FIGURE 5-3 C++ CLASS INTERFACE DEPENDENCE CHARACTERISTIC TO MEASURE RELATIONSHIP (CHARMER) DIAGRAM 138
FIGURE 5-4 C++ CLASS EXTERNAL RELATIONSHIPS CHARMER DIAGRAM 141
FIGURE 5-5 C++ CLASS CONNECTION OBSCURITY CHARMER DIAGRAM 144
FIGURE 5-6 C++ CLASS DEPENDENCY CHARMER DIAGRAM 147
FIGURE 5-7 C++ OBJECT INTERFACE DEPENDENCE CHARMER DIAGRAM 153
FIGURE 5-8 C++ OBJECT EXTERNAL RELATIONSHIPS CHARMER DIAGRAM 156
FIGURE 5-9 C++ OBJECT CONNECTION OBSCURITY CHARMER DIAGRAM - PART 1 161
FIGURE 5-10 C++ OBJECT CONNECTION OBSCURITY CHARMER DIAGRAM - PART 2 162
FIGURE 5-11 C++ OBJECT DEPENDENCY CHARMER DIAGRAM 166

FIGURE 5-12 EXAMPLE OF CONTENT VALIDATION OF OBJECT DEPENDENCY MEASURES USING CHARMER DIAGRAM 171
FIGURE 6-1 MEASUREMENT INSTRUMENT IMPLEMENTATION STAGE OF THE MEASURE DEVELOPMENT PROCESS 175
FIGURE 6-2 ELEMENTS OF A SOFTWARE BASED MEASUREMENT INSTRUMENT 177
FIGURE 6-3 MEASUREMENT INSTRUMENT IMPLEMENTING C++ CLASS AND OBJECT MODULARITY MEASURES 182
FIGURE 6-4 BASIC SOFTWARE MATHEMATICAL MODEL 186
FIGURE 6-5 IMPLEMENTATION OF BASIC SOFTWARE MODEL 187
FIGURE 6-6 UPDATE OF FIGURE 5-6 C++ CLASS EXTERNAL RELATIONSHIPS CHARMER DIAGRAM, REFLECTING MEASUREMENT INSTRUMENT IMPLEMENTATION 195
FIGURE 6-7 UPDATE OF FIGURE 5-10 C++ OBJECT EXTERNAL RELATIONSHIPS CHARMER DIAGRAM, REFLECTING MEASUREMENT INSTRUMENT IMPLEMENTATION 197
FIGURE 6-8 UPDATE OF FIGURE 5-11 C++ OBJECT CONNECTION OBSCURITY CHARMER DIAGRAM, REFLECTING MEASUREMENT INSTRUMENT IMPLEMENTATION 198
FIGURE 6-9 UPDATE OF FIGURE 5-12 C++ OBJECT CONNECTION OBSCURITY CHARMER DIAGRAM, REFLECTING MEASUREMENT INSTRUMENT IMPLEMENTATION 199
FIGURE 6-10 UPDATE OF FIGURE 5-13 C++ OBJECT DEPENDENCY CHARMER DIAGRAM, REFLECTING MEASUREMENT INSTRUMENT IMPLEMENTATION 200
FIGURE 7-1 CHARMER DIAGRAM SHOWING CONTENT VALIDATION FOR EMULEPLUS MEASURES DESCRIBING OBJECT CONNECTION OBSCURITY. 220
FIGURE 7-2 CHARMER DIAGRAM SHOWING CONTENT VALIDATION FOR EMULEPLUS MEASURES DESCRIBING OBJECT DEPENDENCY. 222
FIGURE 7-3 EFFECT OF MEASURE CIS5 ON EMULEPLUS UNWEIGHTED CLASS MODULARITY AGGREGATE VALUES 227
FIGURE 7-4 EFFECT OF MEASURE CIS5 ON EMULEPLUS WEIGHTED CLASS MODULARITY AGGREGATE VALUES 227
FIGURE 7-5 EFFECT OF MEASURE CIS5 ON EMULEPLUS CLASS MODULARITY DISTANCE VALUES 228
FIGURE 7-6 COMPARISON OF WEIGHTED AND UNWEIGHTED MODULARITY AGGREGATE AND MODULARITY DISTANCE INDICATORS OF EMULEPLUS CLASS MODULARITY 229
FIGURE 7-7 EMULEPLUS CLASS LINES OF CODE AND UNWEIGHTED MODULARITY AGGREGATE OUTLIERS 231
FIGURE 7-8 EMULEPLUS CLASS LINES OF CODE AND WEIGHTED MODULARITY AGGREGATE OUTLIERS 232
FIGURE 7-9 EMULEPLUS CLASS LINES OF CODE AND MODULARITY EUCLIDEAN DISTANCE OUTLIERS 232
FIGURE 7-10 FREQUENCY HISTOGRAM OF EMULEPLUS CLASS MODULARITY NORMALISED AGGREGATE VALUES 240
FIGURE 7-11 FREQUENCY HISTOGRAM OF EMULEPLUS OBJECT MODULARITY AGGREGATES 243
FIGURE 7-12 REPRESENTATION OF AN OBJECT ORIENTED MODULE WITH LOW INTERFACE DEPENDENCE 249
FIGURE 7-13 GRAPHICAL REPRESENTATION OF CPARTFILE CLASS INTERFACE DEPENDENCE 251
FIGURE 7-14 CLASS CPARTFILE INTERFACE METHODS WITH MORE THAN 32 LINES OF CODE 254
FIGURE 7-15 CLASS CPARTFILE INTERFACE METHOD INVOCATIONS AND ATTRIBUTE ACCESSES 255
FIGURE 7-16 CLASS CPARTFILE INTERFACE METHODS THAT POTENTIALLY INCREASE INTERFACE DEPENDENCE 256
FIGURE 7-17 OBJECT CPARTFILE INHERITS ELEMENTS FROM THREE ANCESTOR CLASSES 258
FIGURE 7-18 GRAPHICAL REPRESENTATION OF CPARTFILE OBJECT INTERFACE DEPENDENCE 259
FIGURE 7-19 OBJECT CPARTFILE INTERFACE METHODS WITH MORE THAN28 LINES OF CODE 262
FIGURE 7-20 OBJECT CPARTFILE INTERFACE METHOD INVOCATIONS AND ATTRIBUTE ACCESSES 263
FIGURE 7-21 OBJECT CPARTFILE INTERFACE METHODS THAT POTENTIALLY INCREASE INTERFACE DEPENDENCE 264
FIGURE APPENDIX 4-1 EMULEPLUS CLASS INTERFACE DEPENDENCE CHARMER DIAGRAM CONTENT VALIDATION 308
FIGURE APPENDIX 4-2 EMULEPLUS CLASS EXTERNAL RELATIONSHIPS CHARMER DIAGRAM CONTENT VALIDATION 309
FIGURE APPENDIX 4-3 EMULEPLUS CLASS CONNECTION OBSCURITY CHARMER DIAGRAM CONTENT VALIDATION 310
FIGURE APPENDIX 4-4 EMULEPLUS CLASS DEPENDENCY CHARMER DIAGRAM CONTENT VALIDATION 311
FIGURE APPENDIX 4-5 EMULEPLUS OBJECT INTERFACE DEPENDENCE CHARMER DIAGRAM CONTENT VALIDATION 312
FIGURE APPENDIX 4-6 EMULEPLUS OBJECT EXTERNAL RELATIONSHIPS CHARMER DIAGRAM CONTENT VALIDATION 313

FIGURE APPENDIX 4-7 EMULEPLUS OBJECT CONNECTION OBSCURITY CHARMER DIAGRAM CONTENT VALIDATION - PART 1 314
FIGURE APPENDIX 4-8 EMULEPLUS OBJECT CONNECTION OBSCURITY CHARMER DIAGRAM CONTENT VALIDATION - PART 2 315
FIGURE APPENDIX 4-9 EMULEPLUS OBJECT DEPENDENCY CHARMER DIAGRAM CONTENT VALIDATION 316

List of Tables

TABLE 4-1 CLASS MODULARITY SUB-CHARACTERISTICS SHARING COMMON NATURAL LANGUAGE MODEL FEATURES 95
TABLE 4-2 RULES FOR THE ASSIGNMENT OF C++ OBJECT-CLASS ELEMENT LEVELS OF PROTECTION 98
TABLE 4-3 SHORTCOMINGS OF C++ OBJECT MODULARITY MATHEMATICAL MODELS 120
TABLE 4-4 OBJECT MODULARITY SUB-CHARACTERISTICS SHARING COMMON NATURAL LANGUAGE MODEL FEATURES 122
TABLE 5-1 MEASURES OF C++ CLASS INTERFACE ELEMENT INTERDEPENDENCE 134
TABLE 5-2 MEASURES OF C++ CLASS INTERFACE ELEMENT INTERDEPENDENCE (CONT.) 135
TABLE 5-3 MEASURES OF C++ CLASS INTERFACE SIZE 136
TABLE 5-4 MEASURES OF C++ CLASS DATA EXPOSURE 137
TABLE 5-5 MEASURES OF CLASS EXTERNAL RELATIONSHIPS 139
TABLE 5-6 MEASURES OF CLASS EXTERNAL RELATIONSHIPS (CONT.) 140
TABLE 5-7 MEASURES OF CLASS UNSTATED RELATIONSHIP 142
TABLE 5-8 MEASURES OF CLASS DISTANT CONNECTION 142
TABLE 5-9 MEASURES OF CLASS UNEXPECTED RELATIONSHIP 143
TABLE 5-10 MEASURES OF CLASS CONNECTION VIA NON-STANDARD INTERFACE 143
TABLE 5-11 MEASURES OF CLASS SERVICE INVOCATION 145
TABLE 5-12 MEASURES OF CLASS EXTERNAL VARIABLE READING 145
TABLE 5-13 MEASURES OF CLASS EXTERNAL FUNCTION WRITING 146
TABLE 5-14 MEASURES QUANTIFYING FEATURES COMMON TO SEVERAL CLASS MODULARITY SUB-CHARACTERISTICS 148
TABLE 5-15 MEASURES OF OBJECT INTERFACE ELEMENT INTERDEPENDENCE 149
TABLE 5-16 MEASURES OF OBJECT INTERFACE ELEMENT INTERDEPENDENCE (CONT.) 150
TABLE 5-17 MEASURES OF OBJECT INTERFACE SIZE 151
TABLE 5-18 MEASURES OF OBJECT DATA EXPOSURE 152
TABLE 5-19 MEASURES OF OBJECT EXTERNAL RELATIONSHIPS 154
TABLE 5-20 MEASURES OF OBJECT EXTERNAL RELATIONSHIPS (CONT.) 155
TABLE 5-21 MEASURES OF OBJECT VARIABLE CONNECTION 157
TABLE 5-22 MEASURES OF OBJECT UNSTATED RELATIONSHIP 157
TABLE 5-23 MEASURES OF OBJECT DISTANT CONNECTION 158
TABLE 5-24 MEASURES OF OBJECT UNEXPECTED RELATIONSHIP 158
TABLE 5-25 MEASURES OF OBJECT UNEXPECTED RELATIONSHIP (CONT.) 159
TABLE 5-26 MEASURES OF OBJECT CONNECTION VIA NON-STANDARD INTERFACE 159
TABLE 5-27 MEASURES OF OBJECT CONNECTION VIA NON-STANDARD INTERFACE (CONT.) 160
TABLE 5-28 MEASURES OF OBJECT SERVICE INVOCATION 163
TABLE 5-29 MEASURES OF OBJECT INTERFACE PROVISION 163
TABLE 5-30 MEASURES OF OBJECT EXTERNAL VARIABLE READING 164
TABLE 5-31 MEASURES OF OBJECT EXTERNAL FUNCTION WRITING 165
TABLE 5-32 MEASURES QUANTIFYING FEATURES COMMON TO SEVERAL OBJECT MODULARITY SUB-CHARACTERISTICS 167
TABLE 6-1 C++ SOFTWARE MODULARITY MEASUREMENT MODEL SETS DERIVED FROM C++ BASIC SOFTWARE MODEL SOURCE SETS 190
TABLE 6-2 IMPLEMENTATION OF C++ MATHEMATICAL ENTITY MODELS AS C++ SOFTWARE MODULARITY MEASUREMENT MODEL 191
TABLE 6-3 MEASURES OF C++ OBJECT MODULARITY OMITTED FROM MEASUREMENT INSTRUMENT IMPLEMENTATION 194
TABLE 6-4 CHARACTERISTIC TO MEASURE RELATIONSHIP (CHARMER) DIAGRAMS DESCRIBING IMPLEMENTED MEASURES 202
TABLE 7-1 RESULTS OF EMULEPLUS SYSTEM CONTENT VALIDATION 223
TABLE 7-2 EMULEPLUS CLASS LINES OF CODE, MODULARITY AGGREGATE AND MODULARITY DISTANCE DISTRIBUTION STATISTICS 230
TABLE 7-3 STRENGTH OF RELATIONSHIPS BETWEEN EMULEPLUS CLASS LINES OF CODE, UNWEIGHTED AND WEIGHTED MODULARITY AGGREGATES AND MODULARITY DISTANCE 233
TABLE 7-4 EMULEPLUS SYSTEM CLASS MODULARITY AGGREGATE STATISTICS 239
TABLE 7-5 EMULEPLUS MODULARITY AGGREGATES OF CLASSES WITH RELATIVELY LOW MODULARITY 241
TABLE 7-6 EMULEPLUS SYSTEM OBJECT MODULARITY AGGREGATE STATISTICS 242

TABLE 7-7 EMULEPLUS NORMALISED MODULARITY AGGREGATES OF OBJECT-CLASSES WITH POTENTIALLY LOW MODULARITY 245
TABLE 7-8 SUMMARY OF CLASS CPARTFILE INTERFACE SIZE MEASUREMENT DATA 253
TABLE 7-9 SUMMARY OF OBJECT CPARTFILE INTERFACE SIZE MEASUREMENT DATA 261
TABLE APPENDIX 3-1 EMULEPLUS SOFTWARE SYSTEM CLASS MODULARITY MEASURE TO CLASS LINES OF CODE CORRELATION 304
TABLE APPENDIX 4-1 EMULEPLUS SOFTWARE SYSTEM CONTENT VALIDATION MEASURED VALUES 317
TABLE APPENDIX 5-1 EMULEPLUS SOFTWARE SYSTEM CLASS LINES OF CODE, UNWEIGHTED AND WEIGHTED MODULARITY AGGREGATE AND MODULARITY EUCLIDEAN DISTANCE VALUES 322
TABLE APPENDIX 5-2 EMULEPLUS SOFTWARE SYSTEM CLASS AND OBJECT MODULARITY AGGREGATE VALUES 328

Abstract

Software measurement has been of interest to software engineers for almost as long as software has been developed. While the evolution of systematic processes of software development has seen a trend away from reliance on the expertise of individual software developers alone to ensure software quality, systematic processes of software measure development have not evolved to a similar extent. The problem with defining software measures according to an informal process is that the quality of measures can be highly dependent on the expertise of the individual measure developers. If a systematic process of software measure development were defined, that promoted transparency and objectivity in measure development, then this systematic process could support the development of high quality measures by less expert users.

In this thesis, a systematic process of software descriptive measure development is described and demonstrated. The approach taken to defining this systematic process is to investigate the various processes by which currently available software descriptive measures have been developed. These processes are then amalgamated with an established systematic method of measure development used in the field of social science. Applying the stages of measure development thus identified to the task of developing measures to describe C++ class and object modularity tests the feasibility of this measure development process. Insights gained through this testing provide feedback to further refine the process. In this way, a systematic process of descriptive software measure development is defined alongside the definition of a set of measures that provide a detailed description of the complex software characteristic of modularity. The products of each stage of this measure development process assist a user to validate the measures with respect to an intended application, and to analyse and interpret the measurement data obtained by applying the measures to a software system. This is demonstrated in a case study that also provides an indirect indication of the quality of the process by which the measures were developed.

The major contribution of this work is the systematic process of descriptive software measure development, as it has a wide application and can be used to develop measures to describe many software characteristics of interest. A second important contribution is made by the set of measures of C++ class and object modularity developed to demonstrate this systematic descriptive measure development process.

1. Introduction

Software measurement has been of interest to software engineers for almost as long as software has been developed. While the evolution of systematic processes of software development has seen a trend away from reliance on the expertise of individual software developers alone to ensure software quality, systematic processes of software measure development have not evolved to a similar extent. "Software measurement is currently in a phase in which terminology, principles, and methods are still being defined and consolidated." (Briand, Morasca & Basili, 2002, p. 1106) Intuition is still advocated as an acceptable guide to software measure development because "the definition of a measure is itself a very human-intensive activity, which cannot be described and analyzed in a fully formal way" (Briand, Morasca & Basili 2002, p. 1107). The problem with defining software measures according to an informal process is that the quality of measures can be highly dependent on the expertise of the individual measure developers. If a systematic process of software measure development were defined, that promoted transparency and objectivity in measure development, then this process could support the development of high quality measures by less expert users.

In this thesis, a systematic process of software descriptive measure development is described and demonstrated. The major contribution of this work is this process, as it has a wide application and can be used to develop measures to describe many software characteristics of interest. A second important contribution is made by the set of measures developed to demonstrate the measure development process. These measures provide a detailed description of C++ class and object modularity. The systematic process by which they are developed produces not only a set of measure definitions, but also provides a direct link to the theoretical basis from which the measures were developed. This information helps a user of the measures analyse and interpret the measurement data to gain an understanding of the levels of modularity present in a software system.

Measures can serve a descriptive, predictive or prescriptive role (Davis & Hersh 1986). For example, Chidamber, Kemerer (1994) have developed a set of measures that describe object oriented software design complexity. These descriptive measures have been used as predictors of "variations in productivity, rework effort and design effort" (Chidamber, Darcy & Kemerer 1998, p. 633).

Establishing these predictive relationships then allows the same descriptive measures to be used to prescribe "more informed design and resource allocation decisions" (Chidamber, Darcy & Kemerer 1998, p. 633). Fundamental to the predictive and prescriptive roles of measurement is the definition of measures to provide an adequate description of a characteristic of interest.

The process of measure development presented and demonstrated in this thesis is intended for the development of software product measures that directly describe quality characteristics related to the structure of the software. The identification of which software product characteristics to measure and the practical application of such product measures will take place within the wider context of the goals of a measurement program encompassing business measures and process measures as well as product measures (Offen and Jeffery 1997, p. 49). Within this broader context, it is important that fundamental measures, such as those taken directly from software products, provide an accurate description of the product since this information, suitable refined, may form part of the measured description at a process or business context level from which business decisions are made.

1.1. Approach taken to systematic measure development process definition

The approach taken in this thesis to defining a systematic process of descriptive measure development is to investigate the various processes by which some currently available software descriptive measures have been developed. These processes are then amalgamated with an established method of measure development used in the field of social science. Applying the stages of measure development thus identified to the task of developing measures to describe C++ class and object modularity tests the feasibility of this measure development process. Insights gained through this testing provide feedback to further refine the process. In this way, a systematic process of descriptive software measure development is defined alongside the definition of a set of measures that provide a detailed description of a complex software characteristic. The quality of the description of the software that can be obtained from these measures provides an indirect indication of the quality of the process by which the measures were developed. While beyond the scope of this thesis, the usefulness of the measure development process could be further demonstrated by the development of measures to describe other complex software characteristics. Insights gained from this activity could be used to further improve the measure development process.

In the current literature, several different terminologies have been used to describe software measure development. To understand the remainder of this thesis, it is essential that several key concepts be clearly defined and understood. The following section defines the terms that will be used to refer to these key concepts and describes important relationships between these concepts.

1.2. Key concepts

The following points define the terms 'entity', 'characteristic', 'feature' and ‘theory’, as they will be used in this thesis.

- Entity - Different authors have used several different terms to identify "the thing that is going to be measured". In this thesis, in a manner similar to Fenton (1995, p. 2) and Kitchenham (1996, p. 63), the term "entity" will be used to mean the thing that is being measured.
- Characteristic - In this thesis, it is a "characteristic" of the entity that will be described by the measures. Others to use this term in the same way are Henderson-Sellers (1996) and Abreu & Melo (1996). Fenton (1995) and Chidamber and Kemerer (1994) are amongst those to use the term "attribute" to mean this, however the term attribute will later be used to refer to the data elements of an object and class.
- Feature - The term "feature" will be used to describe the structural aspects of the entity that affect the level of a characteristic present.
- Theory – “supposition or system of ideas explaining something, esp. one based on general principles independent of the particular things to be explained” (The Australian Pocket Oxford Dictionary 2002)

Figure 1-1 illustrates the relationships between entities, characteristics, features and descriptive measures.

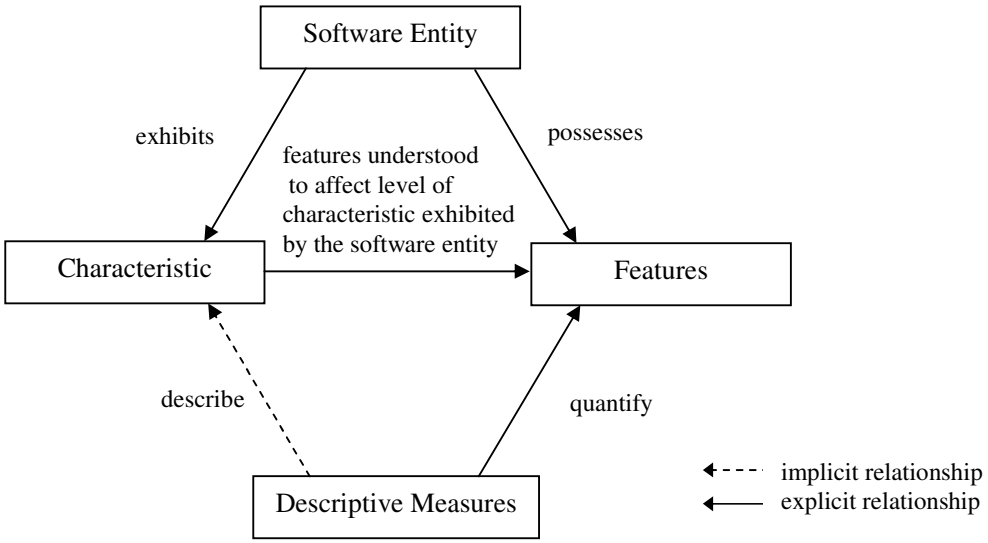


Figure 1-1 Relationships between entities, characteristics, features and descriptive measures

The software entity possesses features that are understood to affect the levels of characteristics exhibited by the software. By quantifying these features, the software measures are able to describe the levels of characteristic present in the software. The explicit relationships between the measures and the features they quantify, and between the features and the characteristic of interest, establish the implicit relationship between the measures and the characteristic they describe.

Figure 1-2 shows an example of a source code entity and the way in which the lines of code feature can be quantified by a measure to provide a description of the size of the source code.

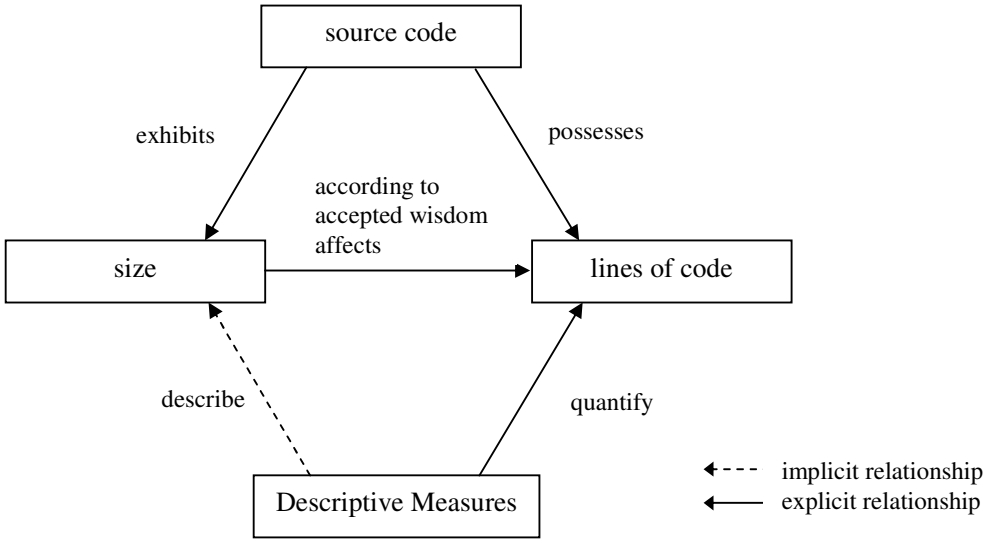


Figure 1-2 Relationships between a source code entity and descriptive measures that quantify lines of code to describe source code size

The first step in establishing the measurement relationships shown in Figure 1-2 is to identify the entity to be measured (source code) and the characteristic to be described by the measures (size). Next, the features of the source code that affect its size must be identified. In this case, lines of code are identified as affecting the size of the source code. The theoretical basis for this selection must be specified so that the descriptive measures can be correctly defined and later interpreted. In this case, the theoretical basis is the accepted wisdom of the software engineering field that says that lines of code affect the size of the source code and that increasing lines of code increases source code size. Once this theoretical basis is established, measures can be defined to quantify the lines of code feature of the source code. When interpreted with respect to the stated theoretical basis from which they were developed, these measures provide a description of the size of source code.

A systematic process of descriptive measure development must support the identification and documentation of the explicit relationships between entities, characteristics, features and descriptive measures. In this way, the implicit descriptive relationship between measures and characteristics is established. The example illustrated in Figure 1-2 describes the development

of a single measure to describe a relatively simple software characteristic with a widely recognised theoretical basis. This thesis describes and demonstrates a process suitable for the development of measures to provide a detailed description of a complex software characteristic that may not be widely understood. This process supports the identification and documentation of the descriptive measure elements and relationships illustrated in Figure 1-1. It also supports the implementation of the defined measures within a measurement instrument, the validation of these implemented measures and the analysis and interpretation of measurement data obtained by applying the measures to a software system.

1.3. Measure development process

The staring point for the development of this systematic software descriptive measure development process is a measure development process commonly used in the social science field (Diamantopoulos & Schlegelmilch 1997) combined with elements of the processes used to develop some of the currently available descriptive software measures. The social science measure development process is applicable to software measure development because it is intended to support the definition and validation of measures describing complex characteristics that are not always well understood. To support the development of such measures, in both fields there is a recognised need to clearly define the characteristic to be described by the measures, to operationally define the measures in an unambiguous way, and to validate the measures to show they provide an adequate description of the characteristics of interest. In this study, through the development of measures to describe C++ class and object modularity, the initial measure development process evolved into a process tailored to the specific needs of software descriptive measure development. Figure 1-3 illustrates the process obtained in this way. This measure development process will be described and demonstrated in this thesis.

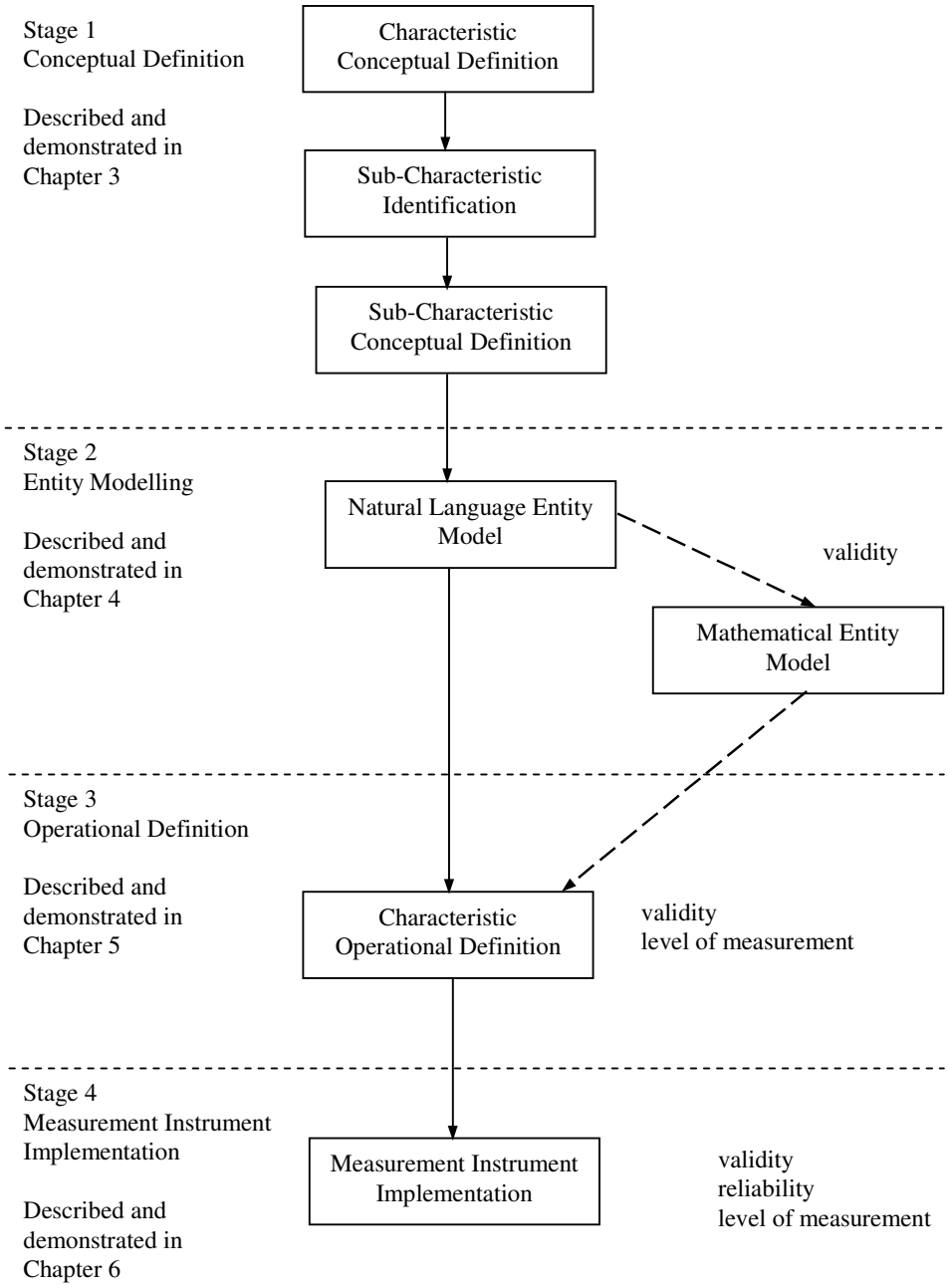


Figure 1-3 Systematic process of software descriptive measure development

Stage 1 is the conceptual definition stage in which the characteristic to be described by the measure is defined "in terms of other concepts, the meaning of which is assumed to be more familiar to the reader." (Diamantopoulos & Schlegelmilch 1997, p. 21). The conceptual

definitions "guide the development" (Diamantopoulos & Schlegelmilch 1997, p. 22) of the operational measure definitions. If the characteristic is complex, it may not be possible to conceptually define it in simple, familiar terms. In this case, the characteristic must be initially defined in terms of the sub-characteristics that contribute to its manifestation in the software. These sub-characteristics can then be conceptually defined. The idea of sub-characterising complex software characteristics before measuring them is not new. Kitchenham, Pfleeger and Fenton (1995, p. 942) recommend that "If you are concerned about a multi-dimensional attribute such as complexity or quality, use different measures for different aspects of the attribute." Meyer recognises that modularity is a complex characteristic of software, stating that "a single definition of modularity would be insufficient; as with software quality, we must look at modularity from more than one viewpoint." (Meyer 1997, p. 39). Each viewpoint could identify a modularity sub-characteristic, which could be conceptually defined and separately measured to provide a description of software modularity. The inability to sub-characterise and precisely define a complex characteristic may indicate that it is not sufficiently well understood and that a better understanding is needed before measures can be developed to describe it.

Stage 2 is the entity modelling stage. This stage is not explicitly included in the social science measure development process; however it is recognised as an important component of software measure development. Pfleeger, Jeffery, Curtis and Kitchenham (1997, p. 36) recommend that software measure developers should "develop more accurate models on which to base better measures". As Figure 1-3 shows, two types of entity models; natural language and mathematical, are defined in this measure development process. The natural language model is fundamental to the development of detailed descriptive measures. Its definition is based on an understanding of the characteristic to be described by the measures, and the ways in which it is manifest in the software. "The most common form of theory within software metrics is that which is required to link direct measurement with phenomenon." (Shepperd & Ince 1993, p. 63). The natural language model expresses the theoretical basis from which the descriptive measures are developed by describing the link between the characteristic (phenomenon) of interest and the software features quantified by direct measurement to describe the levels of characteristic present in the software. Where a software phenomenon is sufficiently well understood, the real-world "experience", "belief", "invention", "hearsay", "practice", "development" and "modelling" (Jeffery & Scott 2002, p. 543) regarding the phenomenon can be transformed into a tentative theory. This tentative theory can then form the basis from which the software features affecting the levels of characteristic present are identified and described in the natural language entity model. The depth of understanding inherent in the selected

theoretical basis of measure development will prescribe the level of detail within the associated natural language model of the software. The detail of the natural language model description of the software will in turn limit the detail of description obtained from measures developed from this natural language model.

The mathematical entity model is optional, as indicated in Figure 1-3 by the broken line connecting it to the other elements of the measure development process. This model describes the features of the software identified in the natural language model as affecting the levels of characteristic present in the software. Mathematical entity models have been widely used to support software measure definition. Churcher and Shepperd (1995b) define an entity-relationship type model to support the definition of object oriented software measures. Briand, Daly and Wust (1999) define a set based model of object oriented software to describe software features of class and object coupling. Mathematical models can provide a precise description of the software and where the type of model is suitable, software measures can be mathematically defined in terms of the model elements. A disadvantage of mathematical entity modelling is that it may not be possible to describe within a particular type of mathematical model, all the software features affecting a characteristic. This means that measures to quantify the missing features cannot be defined in terms of the selected mathematical model. Care must be taken to "choose the model that most clearly emphasises the attribute(s) in question" (Fenton & Melton 1990, p. 178). Stage 2 entity modelling is described and demonstrated in Chapter 4 of this thesis.

Stage 3 of the systematic descriptive measure development process is operational definition. The operational definition defines a characteristic in the way it is to be measured. In social science measurement, conceptual definitions guide the development of the operational definitions and a single characteristic may be operationally defined in several different ways (Diamantopoulos & Schlegelmilch 1997, p. 22). It is recommended that the operational definition "Define the variables so precisely that anyone reading the operational definition could measure the variables in exactly the same way you measured them." (Sproull 1995, p. 34). In the systematic process of software measure development illustrated in Figure 1-3, a characteristic is operationally defined by the measures that quantify the features of the software identified in the natural language model as affecting the level of characteristic present in the software. Ideally, each identified software feature is quantified by at least one measure. The measures can be defined in natural language terms and, where a suitable mathematical model has been defined, the measures can be defined in terms of this model. As Figure 1-3 shows, the

level of measurement achieved by each measure should be determined and stated for each measure. Figure 1-3 also shows that validation is part of the operational definition stage of measure development. This particular validation assesses the degree to which the defined measures provide an adequate description of the features identified in the natural language characteristic model. Stage 3 operational measure definition and validation is described and demonstrated in Chapter 5 of this thesis.

Stage 4 of the systematic descriptive measure development process is measurement instrument implementation. In this stage, the measures operationally defined in the previous stage are implemented within a measurement instrument. It is uncommon to specifically discuss measurement instrument implementation as part of the measure development process. Many measure development "approaches stop at the point at which measurement concepts or specific metrics are identified. They do not define how such measures can be collected and stored, nor (in general) do they define how they can be analysed." (Kitchenham, Hughes & Linkman 2001, p. 788). Measurement instrument implementation is included as part of this measure development process because implementing measures that provide a detailed description of a complex software characteristic can be problematic. The large amounts of data required to formulate a detailed measured description of a software characteristic can be difficult to extract from a software document and may require large amounts of storage and complex manipulations. Addressing some of these issues explicitly in the measure development process will help a future user of the defined measures implement them within their own selected measurement instrument. Stage 4 measurement instrument implementation is described and demonstrated in Chapter 6 of this thesis.

Although Figure 1-3 shows the systematic measure development process as being linear, like many development processes, it may be necessary to return to previous stages to clarify or improve on the work conducted at that stage. These reiterations are not shown specifically on Figure 1-3 however they can occur from any stage back to any previous stage. Once changes have been made to a stage, it is necessary to update all the following stages to take into account the changes made.

Having developed a process by which a set of measures can be defined to provide a detailed description of a complex software characteristic, the worth of this process should be examined. Two indicators of the value of the systematic measure development process are the extent to which the process is able to support the development of measures to describe complex software

characteristics and the usefulness of the measures developed in this way. The worth of the measure development process has been partly demonstrated by developing a set of measures to describe the complex software characteristic of modularity. The ability to demonstrate measure validity with respect to a software system, apply the measures to that software system and analyse the resulting data and present "a 'big picture' of what the software is like" (Pfleeger, Jeffery, Curtis & Kitchenham 1997, p. 41) is an indication of the quality of the set of measures and an indirect indication of the value of the process by which the measures were developed. Chapter 7 of this thesis presents a case study application of the descriptive measures of C++ class and object modularity developed in this thesis according to the systematic process of measure development illustrated in Figure 1-3.

Having defined a set of measures to describe a particular software characteristic, the extent to which these measures represent the characteristic of interest should be determined. Validity and reliability are two properties of measures that provide this information (Carmines & Zeller 1979, p. 11). The following sections briefly describe validity and reliability assessment of measures of complex characteristics. For a more detailed description of this subject, 'Reliability and Validity Assessment' by Carmines and Zeller (1979) is recommended reading.

1.4. Measure validity

Validity is "the extent to which any measuring instrument measures what it is intended to measure." (Carmines & Zeller 1979, p. 17). Three main types of measure validity are content validity, criterion validity and construct validity. Content validity is an estimate of the degree to which a measure or set of measures capture the characteristic of interest (Diamantopoulos & Schlegelmilch 1997, p. 34). Criterion validity is an estimate of "the extent to which a measure can be used to predict an individual's score on some other characteristic" (Diamantopoulos & Schlegelmilch 1997, p. 35). Finally, construct validity is an estimate of the degree to which a measure conforms to theoretically expected behaviour (Sproull 1995, p. 81) (Diamantopoulos & Schlegelmilch 1997, p. 35).

Figure 1-4 is an elaboration of the Figure 1-1 representation of the relationships between entities, characteristics, features and descriptive measures. It shows that the description of a characteristic provided by a set of measures represents an implicit relationship established by the explicit relationships between descriptive measures and features and between features and the characteristic. Validation aims to establish the adequacy of the implicit relationship

between descriptive measures and the characteristic they describe. As Figure 1-4 shows, there are two possible paths to follow to establish this relationship: via the implicit path from measures to characteristic, or via the explicit path from descriptive measures to features and then on to the characteristic.

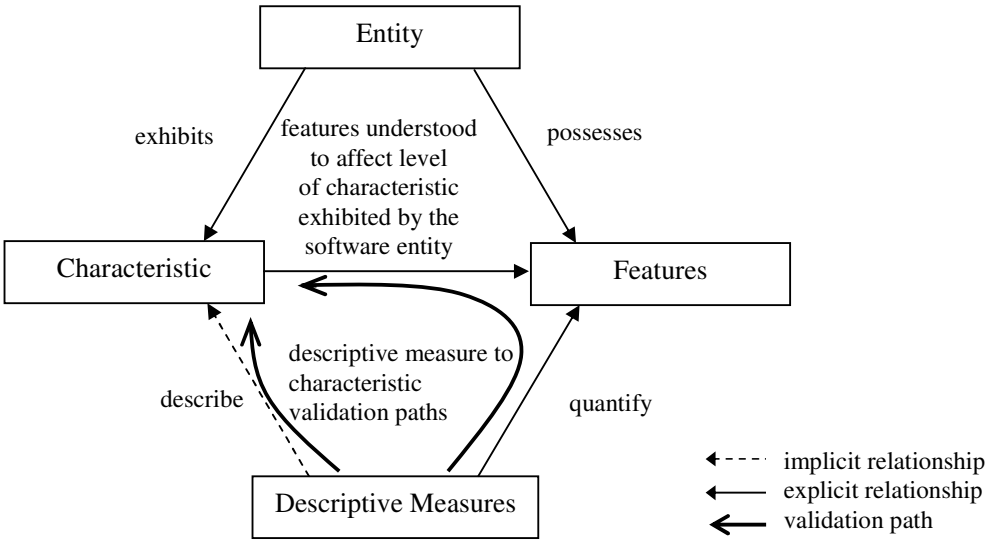


Figure 1-4 Paths of descriptive measure validation

Chidamber and Kemerer (1994) provide an example of a measure validation via the implicit path from descriptive measure to characteristic. Their measures of object oriented software complexity are validated with respect to Weyuker's (1988) set of mathematical properties of complexity measures. Briand, Morasca and Basili (1996) propose a similar set of properties of software size, complexity, cohesion and coupling to "identify and clarify the essential properties behind these concepts" (Briand, Morasca & Basili 1996, p. 69) and use them to "theoretically validate … a family of measures for cohesion and coupling of high-level object-based software designs." (Briand, Morasca & Basili 1999, p. 722). Recently, Arisholm, Briand and Foyen (2004) use some of these same properties to define and validate a set of measures of object oriented software dynamic coupling. These properties establish a direct link between descriptive measures and the characteristic they describe. The strength of this link is dependent on the precision with which the properties are able to "characterize" (Briand, Morasca & Basili 1999, p. 724) the characteristic. They recommend that their "properties are to be interpreted as necessary, but not sufficient." (Briand, Morasca & Basili 1999, p. 724), indicating that the link

from measures to characteristic that they establish may be incomplete. They make the point that the mathematical properties can be used to overcome the lack of conceptual definitions of software characteristics.

> The introduction of appropriate measures is facilitated by the availability of precise definitions for the attributes of interest. Unfortunately, such attributes, e.g., size, complexity, cohesion, coupling, are hardly ever defined in a precise and unambiguous way, if they are defined at all. However, approaches have appeared in the recent literature to provide these attributes with less fuzzy and ambiguous definitions, using mathematical properties to characterize them (Briand, Morasca and Basili 1999, p. 724).

The use of this type of property based validation in the software measurement field has been criticised, identifying the difficulties of adequately defining abstract characteristics with such properties. This in turn could lead to an incorrect, property based validation of such an abstract characteristic. (Kitchenham & Stell 1997). Ideally, measure validity could be established through more widely accepted and conventional methods such as content, criterion and construct validation.

As Figure 1-4 indicates, there is a second path of validation from descriptive measures, through features to the software characteristic of interest. Validation along this path involves firstly examining the link between the characteristic of interest and the features identified as affecting this characteristic to determine whether or not these features have been correctly identified. Next, the link between these features and the measures that quantify them is examined to determine whether or not sufficient measures have been defined. This is the subjective process of content validation.

1.4.1 Content validation

Figure 1-4 shows that content validation via the explicit path from descriptive measures to features to the characteristic of interest requires a statement of the theoretical basis of the measures describing the software features that affect the levels of characteristic present. The systematic process of measure development presented in this thesis expresses this theoretical basis in the conceptual definitions and natural language entity models. This theoretical basis facilitates a content type validation of the measures developed according to this systematic process.

Unlike the other forms of validation, the level of content validity is not determined using a statistical test but by the "logical process of comparing the components of a variable to items of a measure." (Sproull 1995, p. 79). Since content validity is closely related to the purpose in using the measures, it should be assessed individually for each intended application. Sproull (1995, p. 79) describes the process of estimating the content validity of a social science measurement instrument. Modifying this process to the task of software measure content validity estimation results in the process described in Figure 1-5. Sproull's (1995, p. 79) original description of each step in the content validation process is given in italic script after the description of the corresponding software measure content validation step. Step 5 of Sproull's process has been omitted because it is not relevant to software measure content validation.

Software Descriptive Measure Content Validation

1. Examine the characteristics to be described by the measures and list the software features that affect the level of characteristics present. (*Examine the variables of interest and list the tasks or skills or other characteristics involved).*
2. Add to the list the importance and frequency of occurrence of each of the characteristics and associated features. Importance can mean the relevance of the characteristic to the particular study or the influence the feature has on the overall level of characteristic present or some other criteria. Frequency means the frequency with which the software features occur within the software system to be measured. *(Add to the list the importance (criticality) and frequency of occurrence of each of the tasks or skills.)*
3. Re-examine the list to make sure all the features that are crucial to the characteristic of interest are included, even if they occur infrequently. *(Re-examine the list and make sure that all skills or tasks which are crucial to the variable are included even if they occur infrequently. Add any which may have been omitted.)*
4. Compare each of the characteristics and features on the list to the individual measures within the measure suite to ensure that each crucial characteristic and frequently occurring feature is measured by at least one measure. Usually more measures are included to assess those characteristics that are more important and those features that occur more frequently. *(Compare each of the tasks or skills on the list to the items of the measure to ensure that each crucial and frequently occurring task or skill is measured by at least one item. Usually more items are included to assess those skills which are more important and those which occur frequently, an additional aspect of the representativeness of the measure to the variable being measured.)*

Figure 1-5 Steps of descriptive measure content validation. Adapted from Sproull (1995, p.79).

Performing steps 1, 2 and 3 of the content validation process described in Figure 5-4 involves evaluating the quality of the theoretical basis of the measures. A potential user of the measures must examine the conceptual definition of the characteristic and the natural language entity model of the software, as these constitute the specific expression of the theoretical basis of the measures. If this examination reveals the theoretical basis to be acceptable, then the validity of

the Figure 1-4 link between a characteristic and the features that affect it has been established. If this is the case, then step 4 of the content validation can proceed. Otherwise, if the theoretical basis is judged unacceptable, the measures developed from it must be judged invalid for the user's particular proposed application. Step 4 of the content validation process involves a user comparing the set of measures to the features identified as affecting the characteristic. Ideally, each feature is adequately described by one or more measures, in which case the measures can be judged to provide a sufficiently valid description of the characteristic of interest. Only if a set of measures is judged to be sufficiently valid should software measurement proceed.

1.4.2 Construct validation

While content validation is important, and should be assessed for any proposed application of measures, it is not fully sufficient for assessing the validity of measures. Construct validity is a type of validity applicable to software descriptive measures such as those developed in this thesis. “Fundamentally, construct validity is concerned with the extent to which a particular measure relates to other measures consistent with theoretically derived hypotheses concerning the concepts (or constructs) that are being measured.” (Carmines & Zeller 1979, p. 23). As an example, consider a new measure of the height of a person. A construct validation of this new measure could be performed based on a theoretical understanding of the relationship between the height and weight of adults. This theory would state that taller people would tend to weigh more than smaller people. The relationship between the height and weight characteristics of an adult would not correlate perfectly because exceptions to this rule, such a very tall, thin people, do occur however, for a large sample of the population, a reasonably strong correlation between height and weight data would be expected if the new height measure was indeed a valid description of height. This does not mean that a weight measure may be substituted for a height measure as they are describing different, though related characteristics of a person.

Sproull (1995, p. 82) describes the process of estimating the construct validity of a social science measurement instrument. Figure 1-6 describes the steps of software descriptive measure construct validation, modified by the process described by Sproull (1995, p. 82). Where the step is different to that of Sproull, Sproull's (1995, p. 82) original description of each step in the construct validation process is given in italic script after the description of the corresponding software measure construct validation step.

Software Descriptive Measure Construct Validation

1. Examine the theory associated with the characteristic of interest described by the measures. (*Examine the theory associated with the variable of interest).*
2. Select several other characteristics that theory indicates would differentiate entities with differing amounts of the measured characteristic of interest. *(Select several behaviours which the theory indicates would differentiate subjects with differing amounts of the variable.)*
3. Measure the characteristic of interest for a selected validation software system. (*Administer the instrument measuring the variable of interest to the validity sample and record the scores*)
4. Measure the other characteristics associated with the characteristic of interest for a selected validation software system. *(Gather scores for the validity sample on each of the behaviours selected in step #2.)*
5. Analyse the data using appropriate statistical tests to ascertain if subjects scoring high on the major variable and those scoring low are statistically differentiated on each of the selected criterion variable.
6. Accept evidence of construct validity if each of the statistical tests indicates a significant difference or a significant relationship between high and low scorers on the major variable and the criterion variables. If even one of the hypothesised relationships is not supported statistically, then the instrument cannot be said to evidence construct validity.
7. Examine reasons if construct validity is not supported. Possible reasons include: (1) the theory is incorrect, (2) the instrument was not a valid measure of the variable of interest, or (3) there may have been errors in the administration of the instrument, scoring or analysis of the data.

Figure 1-6 Steps of descriptive measure content validation. Adapted from Sproull (1995, p.79).

The data analysis described in Chapter 7 includes a construct type validation of the modularity measures defined in this thesis.

1.5. Measure Reliability

Reliability "concerns the extent to which an experiment, test, or any measuring procedure yields the same results on repeated trials" (Carmines & Zeller 1979, p. 11). It is "the degree to which an instrument measures the same way each time it is used under the same conditions with the same subjects." (Sproull 1995, p. 74). The measures developed in this thesis are intended to be implemented within a, software based measurement instrument that will automatically scan the software document, collect the required data and generate the measures. By its nature, this instrument will be perfectly reliable and thus reliability estimated will not be performed in this thesis. Where a measurement instrument, such as one based on people collecting data from software documents, introduces random measurement errors, then reliability should be estimated.

1.6. Level of measurement

Measures can be classified according to the level of measurement they achieve (Stevens 1946). The level of measurement achieved by a measure dictates the mathematical manipulations and statistical analysis that can be performed on the collected measure data. The four major levels of measurement, ranked in order from lowest to highest level are nominal, ordinal, interval and ratio (Stevens 1946, p. 678). The maximum level of measurement achieved by a measure should be stated so that the appropriate data analysis techniques can be employed. Many data analysis and measurement texts discuss level of measurement. For example, Sproull (1995, pp. 67-74) gives a detailed discussion of each level of measurement including the arithmetic and statistical operations permitted.

1.7. Comments

The process of measure development illustrated in Figure 1-1 represents the end product of this study rather than the starting point. The development of the measures to describe C++ class and object modularity began with the idea that a detailed sub-characterisation and conceptual definition of object oriented class and object modularity would provide a foundation from which measures to provide a detailed description of modularity could be developed. As measure development proceeded, it became clear that a large number of measures would need to be defined to quantify all the software features identified as affecting the levels of modularity present in the C++ software. The decision was made to continue to develop measures to provide this very detailed description even though implementing such a large set of measures could present problems. Problems were also anticipated when attempting to analyse the large data set resulting from applying the measures to even a small software system. The reason for deciding to carry on with the development of a detailed measure set was that this represents an extreme case that would be most likely to test the measurement development process and highlight any deficiencies. In fact, difficulties were encountered at all the stages of measure development and these are discussed in the relevant chapters of this thesis. The example of how to develop a set of detailed descriptive measures presented in this thesis serves as a guide for anyone deciding to develop a similar set of measures to provide a detailed description of a different complex software characteristic.

The process of measure development illustrated in Figure 1-1 and described and demonstrated in this thesis provides a guide to the development of descriptive software measures. Avenues for future work identified in the final chapter of this thesis, will further test and refine this process.

1.8. Structure of this thesis

This thesis document is organised in the following way.

Chapter 1 - Introduction

Chapter 2 - Literature Review

Chapter 3 - Conceptual Definition stage of measure development

- Section 3.1 - description of conceptual definition stage
- Section 3.2 - demonstration of conceptual definition stage

Chapter 4 - Entity Modelling stage of measure development

- Section 4.1 - description of entity modelling stage
- Section 4.2 - demonstration of entity modelling stage

Chapter 5 - Operational Measure Definition stage of measure development

- Section 5.1 - description of operational definition stage
- Section 5.2 - demonstration of operational measure definition stage

Chapter 6 - Measurement Instrument Implementation stage of measure development

- Section 6.1 - description of instrument implementation stage
- Section 6.2 - demonstration of instrument implementation stage

Chapter 7 - Application of C++ Class and Object Modularity Measures

- Section 7.1 Data analysis - techniques for modularity data reduction
- Section 7.1 Content validation – eMulePlus software system
- Section 7.2 Example of a construct validation – eMulePlus software system
- Section 7.3 - Case Study 1 - modularity of the eMulePlus software system
- Section 7.4 - Case Study 2 - interface dependence of the CPartFile class module

Chapter 8 - Discussion and Conclusion

Appendices 1 to 6

Bibliography

Several appendices are included with this thesis document. The following lists these appendices and the chapters they are associated with.

Appendix 1 - Entity-Relationship Model Set Definitions
Associated with Chapter 4, Chapter 5 and Chapter 6

Appendix 2 - Basic Model to Measurement Model Transformations
Associated with Chapter 6

Appendix 3 – eMulePlus C++ Class Modularity to Lines of Code Correlation Data
Associated with Chapter 7

Appendix 4 - eMulePlus Content Validation
Associated with Chapter 7

Appendix 5 - eMulePlus Modularity Data
Associated with Chapter 7

Appendix 6 - CPartFile Interface Dependence Data
Associated with Chapter 7

2. Literature Review

This chapter reviews literature related to systematic processes of object oriented software descriptive measure development dating from 1988 to 2004. The literature reviewed is selected on the basis of its contributions to the understanding of how descriptive measures can be systematically developed to provide a detailed description of a complex software characteristic. Literature that is focussed more on software process measurement and on the development of measures for function oriented software is not reviewed.

The chapter is organised as follows. The first section surveys some existing processes of software measure development. The next section discusses the processes used to develop some of the currently available object oriented software descriptive measures. Following this is a discussion of the importance of clearly expressing the theoretical basis from which the measures are developed. Finally, the conclusion section restates the research objectives of this thesis and briefly presents the approach to software measure development that will be described and demonstrated in this thesis.

This thesis addresses the issue of how to develop measures to provide a detailed description of a complex software characteristic. This issue needs to be addressed because currently, descriptive software measures are often developed according to various informal processes. These processes are not documented and explained by the measure developers in ways that allow other people to reuse them to develop new measures. They are also directed more to the development of measures that provide a high level, rather than detailed description of software characteristics. This thesis evaluates and builds on these informal processes to define a systematic process of detailed descriptive software measure development.

Briand, Morasca and Basili advocate a disciplined approach to software measure development, believing that this will allow practitioners and researchers to:

> 1. build upon a solid theoretical basis,
> 2. link the measure to the application at hand,
> 3. provide a clearer rationale of the underlying definition of a measure and its applications,
> 4. judge whether it is necessary to define a new measure or instead reuse an existing one for a specific application, and
> 5. interpret the results of an experiment or a case study, especially when one does not obtain the expected results. (Briand, Morasca & Basili 2002, pp. 1106-1107)

These same benefits are expected from the software descriptive measure development process presented in this thesis. Two widely recognised and accepted systematic software measure development process are the IEEE Standard for a Software Quality Metrics Methodology (IEEE Computer Society 1992; IEEE Computer Society 1998) and the Goal/Question/Metric (GQM) paradigm for formalising aspects of the software engineering process model of the TAME (Tailoring a Measurement Environment) project (Basili 1988). The process of measure development described in this thesis is compatible with, and extends these two processes. In general terms, the IEEE and GQM processes provide a high-level description of what must be done, while the measure development process of this thesis provides a detailed description of how to accomplish the transition from identified software characteristic of interest to defined measures that describe the characteristic. The following sections discuss these compatibilities.

2.1. Processes of software measure development

The IEEE Standard for a Software Quality Metrics Methodology (IEEE Computer Society 1992; IEEE Computer Society 1998) describes the steps that should be taken to identify the attributes of an entity that need to be measured to describe high level quality factors. With respect to the IEEE Software Quality Metrics Framework (IEEE Computer Society 1998, p. 4), the process described in this thesis complements the IEEE framework by describing in detail how to accomplish the transition from quality sub-factor to valid and reliable measures that describe the sub-factor. Figure 2-1 illustrates this relationship.

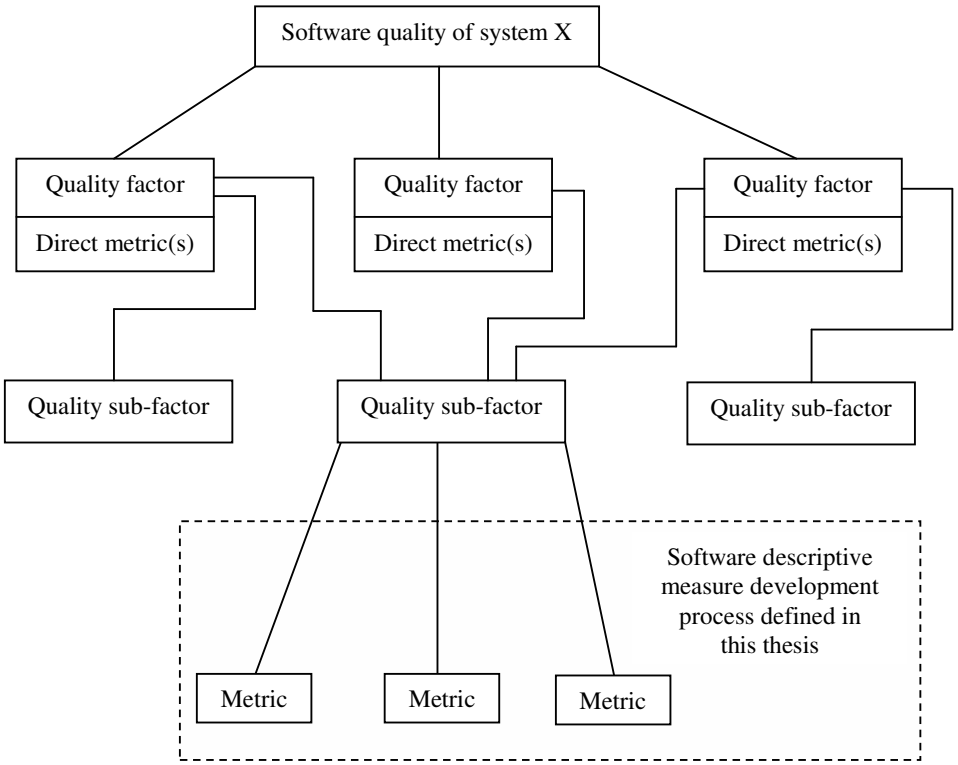


Figure 2-1 Contribution of software descriptive measure development process to software quality metrics framework (IEEE Computer Society 1998, p. 4).

The Goal/Question/Metric (GQM) paradigm (Basili 1988) provides guidelines for the determination of the types of measures that should be made to satisfy the stated goals of a software development process. Figure 2-2 illustrates the way the software descriptive measure development process described and demonstrated in this thesis fits into the GQM paradigm.

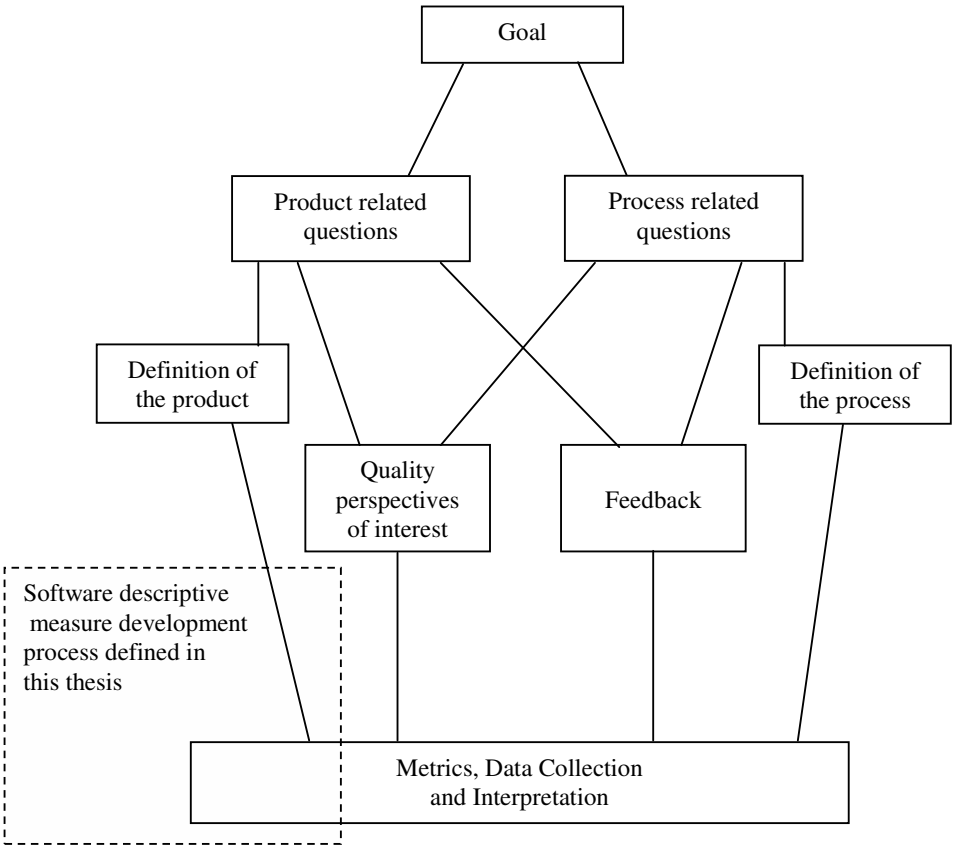


Figure 2-2 Contribution of software descriptive measure development process to TAME GQM Paradigm (Basili 1988).

The results of using the GQM paradigm are a set of software project goals that are "refined into a set of quantifiable questions that specify metrics." (Basili 1988, p. 761). The software measure development process described in this thesis complements the GQM paradigm by providing a guide to the definition, implementation and interpretation of the required product definition metrics.

Recently, Briand, Morasca and Basili (2002) describe a systematic method of software measure development designed to enhance the GQM paradigm (Basili 1988). This method, called the GQM/MEtric DEfinition Approach (GQM/MEDEA), supports the development of predictive software measurement systems. It complements the GQM process by describing a method of defining and validating measures that describe an independent software characteristic so that these measures can then be then used to predict a future characteristic of interest. The main

focus of the GQM/MEDEA process is establishing the prediction relationship between the dependent descriptive measures and the independent characteristic of interest. The measure developer's intuitive understanding of the characteristic of interest provides a theoretical basis from which the independent descriptive measures are developed. This intuitive understanding, along with a proposed set of mathematical properties possessed by measures of a particular characteristic, also provides the basis of descriptive measure validation.

> The measures for the independent attributes may not satisfy the refined properties for the independent attributes. The measure definition process - like any human-intensive activity- is subject to errors and cannot provide 100 percent certainty that correct results are always delivered. It is useful to make sure that the measures for the independent attributes comply with one's formalized intuition. (Briand, Morasca and Basili 2002, p. 1118)

The GQM/MEDEA process of software measure development does not require the explicit expression of the intuitive understanding of the characteristic of interest that provided the theoretical basis of descriptive measure development. This means that a potential user of these measures must rely on their own intuitive understanding to decide whether or not the measures are sufficiently valid for their potential application. A less expert user may not have sufficient knowledge to make this determination. Were the theoretical basis of the measures clearly expressed, a potential user could use this information to determine measure validity and to analyse and interpret the measurement data. Briand, Morasca and Basili (2002, p. 1107) believe that "… the definition of a measure is itself a very human-intensive activity, which cannot be described and analysed in a fully formal way". As will be shown in this thesis, where a software characteristic is sufficiently well understood, it is possible to apply a formal process to the task of developing measures to describe the characteristic of interest.

Kitchenham (1996) recognises the importance of developing software descriptive measures according to specific guidelines and identifies the following principles of measure definition:

> - We need to *specify the entities* to which the measures apply. This is essential to ensure measures are comparable.
> - We need to *define the attribute* we are measuring. This is essential to ensure that we can interpret and use our results.
> - We need to *define the unit* we are using. This is essential to ensure that we understand the values we obtain.
> - We need to *specify the measurement instrument* we are using.
>
> In addition, we may also need to define a *measurement protocol*. (Kitchenham 1996, pp. 63-64)

While it is necessary to observe all these principles when developing software measures, defining the attribute described by the measures is particularly important. As Kitchenham indicates, without the understanding of the attribute conveyed by its definition, a potential user is unable to interpret and use the measured results. In this thesis, as illustrated in Figure 1-3, the definition of the attribute (characteristic) to be described by the measures is the first stage of measure development and guides the remaining measure development stages. Specifying the measurement instrument and protocol used to implement the measures is another important stage of measure development advocated within the previously listed principles of Kitchenham. It is the measures, as they are implemented in measurement instruments, which ultimately provide the description of a particular software characteristic of interest. It is important that one of the products of the measure development process is a description of how the measures should be implemented in a measurement instrument. It is also important to describe the way that a set of measures are actually implemented within a selected measurement instrument so the theoretical and actual implementations can be compared and any shortcomings noted. In this thesis, measurement instrument implementation is the final stage of descriptive measure development process illustrated in Figure 1-3.

Many measures to describe object oriented software are currently available and several are widely recognised within the software development community. The following section discusses the processes by with several of these measures have been developed.

2.2. Processes currently used to develop descriptive software measures

Many of the descriptive software measures that are currently available have been developed according to informal processes that comply with some of Kitchenham's (1996) measure development principles. Elements found in these processes include:

- definition of the attribute to be described by the measures
- natural language modelling of the entity to be measured
- mathematical modelling of the entity to be measured
- natural language measure definition
- mathematical measure definition
- measure validation
- measure implementation

Different groups of software measure researchers have developed measures by performing, with varying degrees of rigour, some or all of these process elements. The following discusses the processes used to develop some of the existing descriptive measures of object oriented software.

The suite of measures defined by Chidamber and Kemerer (1994) are intended to describe the "complexity in the design of classes" (Chidamber & Kemerer 1994, p. 477). Coupling, cohesion, 'complexity of an object' and 'scope of properties' are identified as sub-characteristics of 'complexity in design of classes', although this choice is not justified by reference to an established understanding of this software characteristic. Coupling, cohesion and 'scope of properties' are conceptually defined; 'complexity of an object' is not. Bunge's (1977; 1979) theory of ontology provides the theoretical basis for the identification of software features that affect the levels of these sub-characteristics present in the software. This theoretical basis is expressed as discussions throughout this paper, as 'viewpoints' accompanying each measure definition and as a footnote. Chidamber and Kemerer's Weighted Methods Per Class (WMC), Response For a Class (RFC) and Lack of Cohesion in Methods (LCOM) measures are defined in terms of a simple mathematical model of the software. Their Depth of Inheritance Tree (DIT), Number of Children (NOC) and Coupling Between Object Classes (CBO) measures are defined in natural language terms. These measure definitions have been shown to be ambiguous (Churcher & Shepperd 1995a), motivating Henderson-Sellers (1996) and Hitz and Montazeri (1995) to redefine some of them more precisely. The properties proposed by Weyuker (1988) as necessary for a measure of software complexity to possess are used to demonstrate measure validity. The measures are implemented in two separate measurement instruments, although details of these implementations are not provided. The importance of this study is that it recognised the need to develop measures from a "theoretical base" (Chidamber & Kemerer 1994, p. 476).

The process by which Chidamber and Kemerer's measures were developed could be improved by conceptually defining all the sub-characteristics described by the measures and by more clearly expressing the theoretical basis from which the measures were developed. For example, the theoretical basis of the Weighted Methods per Class (WMC) (Chidamber & Kemerer 1994, p. 482) measure is expressed through the combination of a discussion of Bunge's (1977) view of complexity of an object (Chidamber & Kemerer 1994, p. 479), a statement of the theoretical basis accompanying the WMC measure definition (Chidamber & Kemerer 1994, p. 482) and a footnote to this statement (Chidamber & Kemerer 1994, p. 482). A potential user is only able to

fully understand the theoretical basis of the WMC measure by examining all three of these expressions of theory. If the connection between the measure, the features of the software it quantifies and the aspects of the characteristic described by the measure were more clearly expressed, a user of the measure would be more easily able to apply the measure and interpret the resulting data. A systematic process of descriptive software measure development should incorporate a precise expression of these connections.

Bieman and Kang (1995) define measures to describe object oriented class cohesion. The motivation for their work is to define measures to provide a more detailed description of cohesion than that provided by the LCOM measure (Chidamber & Kemerer 1994, p. 488). Bieman and Kang conceptually define cohesion in general terms as the "relatedness of module components" (Bieman & Kang 1995, p. 259), but it is not further sub-characterised. This means that the conceptual definition does not convey a detailed understanding of cohesion from which to base the development of detailed measures. The theoretical basis of the measures is expressed as a detailed discussion of the authors' understanding of the ways in which relations between class components affect the levels of cohesion present in a class. A simple set based mathematical model is defined to support the natural language model. The measures to describe class cohesion are defined in terms of functions that count the number of elements in the mathematical model sets. The measures are not validated to determine whether or not they adequately describe class cohesion. The type of measurement instrument within which the measures are implemented is specified but no details are provided regarding the degree to which this instrument is able to implement the defined measures. The strengths of Bieman and Kang's study are that the authors recognise the need to define the attribute to be measured, describe the theory behind the measure development, mathematically model the software and define the measures in terms of this mathematical model. This measure development process could be improved by providing a more detailed description of the aspect of cohesion described by each measure and expressing the theoretical basis of the measures more specifically.

Abreu and Carapuca (1994) define a set of Metrics for Object Oriented Design (MOOD) measures to describe internal quality of object oriented design. The stated theoretical basis for the measures is that use of object oriented abstractions increases software internal quality (Abreu & Carapuca 1994, p. 2). Encapsulation and information hiding, inheritance, coupling and clustering, polymorphism, and reuse are identified as sub-characteristics of software quality and are conceptually defined. The theoretical basis, identifying the software features affecting the levels of these sub-characteristics present in the software, is not stated. A

mathematical model of the software is defined to describe these features, and measures are defined in terms of the mathematical model. A subsequent study (Abreu, Goulao & Esteves 1995) provides examples of software code to further explain the measure definitions. These authors have followed the general measure development process in that they express the general theoretical basis for sub-characterisation and provide conceptual definitions of these sub-characteristics. Identification of the software features affecting the levels of sub-characteristics present appears to be based on the authors' own understanding of the software rather than on an established theory. Potential users of these measures must decide, based on their own understanding of these sub-characteristics, whether this feature identification is sufficient and whether or not the measures adequately describe these features.

Briand, Morasca and Basili define a set of measures of object oriented software coupling and cohesion "related to declaration links among data and subroutines appearing in high-level design module interfaces." (Briand, Morasca & Basili 1999, p. 723). A "property-based approach" (Briand, Morasca & Basili 1999, p. 724) is used to characterise an understanding and knowledge of coupling and cohesion. This in turn provides "theoretical support" (Briand, Morasca & Basili 1999, p. 724) for the definition of measures of coupling and cohesion. These same properties are then used to "provide supporting evidence that the measures are theoretically valid" (Briand, Morasca & Basili 1999, p. 724). These properties are intended to compensate for a lack of conceptual definition of coupling and cohesion by providing "these attributes with less fuzzy and ambiguous definitions using mathematical properties to characterize them." (Briand, Morasca & Basili 1996, p. 724). Briand, Morasca and Basili (1999) develop a natural language model of the software, supported by a mathematical model. Measures are defined in terms of this mathematical model. Later, Briand, Daly and Wust (1997; 1999) define a more detailed mathematical software model and define several existing measures of coupling and cohesion more clearly in terms of this model. This general mathematical model clarifies the operational definition of the selected measures and is a strength of this study. A similarly detailed mathematical model would support the definition of the many measures required to provide a detailed description of a complex software characteristic. Where appropriate, the definition of a similar mathematical model should be incorporated into a systematic process of software measure development.

Recently, Arisholm, Briand and Foyen (2004) have developed a set of measures to describe the dynamic coupling of object oriented software as it executes. Coupling is not specifically conceptually defined in this study however, the properties of coupling measures proposed by

Briand, Morasca and Basili (1996) are used to validate the developed measures. This implies that the general definition of coupling as "the amount of relationship between elements belonging to different modules of a system" (Briand, Morasca & Basili 1996, p. 78) associated with these coupling properties is applicable to the developed measures of dynamic coupling. The coupling framework of Briand, Daly and Wust (1999) provides the basis of the identification of the software features to quantify. Measures are defined to describe the strength of both import and export dynamic coupling of object oriented classes and objects by quantifying the dynamic messages, distinct method invocations and distinct classes features of the software (Arisholm, Briand & Foyen 2004, p. 496). These measures "are defined in an informal, intuitive manner but also using a formal framework based on set theory and first-order logic." (Arisholm, Briand & Foyen 2004, p. 492). Defining the measures in terms of a formal framework ensures "that the definitions are precise and unambiguous" (Arisholm, Briand & Foyen 2004, p. 492). This formal framework is a set based mathematical software model. Although similar to the mathematical software model defined by Briand, Daly and Wust (1999), this new model is based on a class diagram specifically describing dynamic coupling (Arisholm, Briand & Foyen 2004, p. 494) rather than on a general model of object oriented software. The measures of dynamic coupling are defined in terms of the set based software model using a form of relational calculus. To determine the degree to which the measures provide an adequate description of coupling, the measures are validated with respect to a set of mathematical properties of coupling measures proposed by Briand, Morasca & Basili (1996, pp. 78-79). The measures are implemented within a software based measurement instrument. A significant strength of the process used to develop these measures is the mathematical software model used to support the definition of the measures. The measures are defined as sets of tuples that satisfy a tuple expression. A similar style of relational calculus measure definition, developed independently of Arisholm, Briand and Foyen (2004), is used to define measures of C++ class and object modularity in this thesis.

The process of measure development employed by Arisholm, Briand and Foyen (2004) could be improved by more clearly presenting the theoretical basis from which the measures were developed. These measures provide a high level description of dynamic coupling and so, are developed based on a high level understanding of coupling. This theoretical basis is presented in an informal way as part of the discussion associated with the measure definitions (Arisholm, Briand & Foyen 2004, pp. 495-497). When developing measures to provide a detailed description of the software, such an informal presentation of theory could be difficult to understand and link to each developed measure. A specific expression of the theory regarding a

software characteristic of interest, linked specifically to the measures describing each aspect of the characteristic will help a potential user validate, apply and interpret the measures that together provide a detailed description of a software characteristic.

From the previously reviewed measure development processes, it can be seen that in general, the theoretical basis of the developed measures is generally not specified in detail. Also, the characteristic to be described by the developed measures is conceptually defined in general rather than specific terms. Natural language modelling of the software to describe the ways in which the characteristic of interest is manifest in the software is often expressed as part of the general discussion of the measures rather than as a specific and identified product of the measure development process. Mathematical modelling of the software entity to be measured is an area of software measure development that has improved from early software measurement projects to become a strength of the most recent measure development project (Arisholm, Briand & Foyen 2004). This in turn has resulted in an improvement of the ways in which software measures are defined. A mathematical software model supports the unambiguous definition of software measures and so, the development of a better mathematical model has in turn resulted in the more precise definition of software measures. One aspect of software mathematical modelling not addressed by currently used software measure development processes is the degree to which the mathematical model is able to describe all the software features that are understood to affect the levels of characteristic of interest present in the software. Pfleeger, Jeffery, Curtis and Kitchenham (1997, p. 36), when discussing the "state of the gap" in software measurement, suggest that software measurement researchers should "consider model validity separate from measure validity, and develop more accurate models on which to base better measures". Validation of the mathematical software model demonstrates the accuracy of the model from which measures are defined. While many of the previously reviewed measure development processes include measurement instrument implementation, the degree to which the selected measurement instrument is able to implement the defined measures with sufficient validity, and reliably collect the measurement data, is not discussed. Measurement instrument validity should be assessed and judged adequate before the instrument is applied to describe a software system. "The notion of validity is not specific to software engineering, and general concepts that we rarely consider - such as construct validity and predictive validity - should be part of any discussion of software engineering measurement." (Pfleeger, Jeffery, Curtis & Kitchenham 1997, p. 36) Different types of validity and styles of validation may be appropriate to software measurement and research is needed to investigate

this possibility. In this thesis, the applicability of content and construct validities (Diamantopoulos & Schlegelmilch 1997; Sproull 1995) to the estimation of descriptive measure validity is investigated.

The process of descriptive software measure development illustrated in Figure 1-3 and described and demonstrated in Chapters 3 to 7 of this thesis supports the development of descriptive measures to provide a detailed description of a complex software characteristic. This process encompasses the explicit statement of the theoretical basis of the measures, mathematical modelling of the software, precise measure definition and measurement instrument implementation. The products of this process support the validation of the measures, as they are implemented in a selected measurement instrument, to determine the degree to which they provide an adequate description of the characteristic of interest.

2.3. Theoretical basis of descriptive software measure development

A detailed understanding of the software characteristic of interest, and the ways in which it is manifest in the software, is essential to the development of descriptive measures of the characteristic. "The most common form of theory within software metrics is that which is required to link direct measurement with phenomenon." (Shepperd & Ince 1993, p. 63). To develop measures that provide a detailed description of a complex software characteristic (or phenomenon), a theory or understanding is required regarding the ways in which the features of the software, quantified by direct measurement, affect the levels of characteristic present in the software. The theoretical basis for the development of descriptive software measures may be "one's own intuitive understanding of the studied phenomena and needs to be explicit so it can be discussed, questioned, and refined." (Briand, Morasca & Basili 2002, p. 1114) Where a "rich discussion of the ideas, and the development of general understandings of the meaning of concepts" (Leaney, Rowe & O'Neill 2002) has occurred, this too can form a basis for measure development. Alternatively, when the "experience", "belief", "invention", "hearsay", "practice", "development" and "modelling" (Jeffery & Scott 2002, p. 543) regarding a software phenomenon has been formulated into a "tentative theory" (Jeffery & Scott 2002, p. 542), and this tentative theory has been evaluated by the software engineering community, it may provide a good basis for software measure development.

To demonstrate the systematic process of software descriptive measure development, a set of measures describing object oriented C++ class and object modularity will be developed in this thesis. Modularity is selected as the characteristic of interest because it is a relatively complex software characteristic that is relatively well understood by the software engineering community. This detailed understanding supports the development of detailed descriptive measures. The development of a theoretical understanding of software modularity can be seen in the work of Stevens, Myers and Constantine (1974) who proposed coupling and cohesion as important sub-characteristics of function oriented software modularity. Coupling and cohesion continue to be regarded as important sub-characteristics of modularity and Briand, Daly and Wust (1997; 1999) list many measures defined by several authors to describe object oriented software coupling and cohesion. The degree to which measures of coupling and cohesion provide an adequate description of object oriented software modularity is examined by Abreu and Goulao (2001). They conclude that in practice, "coupling and cohesion do not seem to be the dominant driving forces when it comes to modularization" (Abreu & Goulao 2001, p. 47). It is possible that "modularization driving forces" other than coupling and cohesion exist in object oriented software systems.

In this thesis, Meyer's (1997, pp. 39-64) theory of object oriented software modularity forms the theoretical basis of the development of descriptive measures of modularity that demonstrates the systematic process of measure development. Meyer's rules of modularity (1988; 1997, pp.46-53) have been previously used as the basis for software measure interpretation. Abreu, Goulao and Esteves (1995, p53) refer to Meyer's (1988) rule of information hiding when discussing the interpretation of their Attribute Hiding Factor (AHF) and Method Hiding Factor (MHF) measures. They also reference Meyer's (1988) rule of small interfaces to support the interpretation of their Coupling Factor (COF) measure (Abreu, Goulao & Esteves 1995, p. 54). In a similar way, Ammann and Cameron (1994 p. 144) refer to Meyer's (1988) rule of weak coupling, which is the same as the later small interfaces rule (Meyer 1997, p. 48), to support the interpretation of their measured results. Briand, Morasca and Basili (1999, p 734) recognise Meyer's (1988) rule of weak coupling as work related to their development of measures of object-oriented software coupling.

Meyer's (1997, pp. 46-53) rules of modularity identify major modularity sub-characteristics and indicate the features of object oriented software that affect the levels of these sub-characteristics present in the software, making them suitable for use as a basis for descriptive measure development. This suitability combined with the long-standing acceptance of these

rules by the software engineering community, supports their selection as the theoretical basis for the measures developed in this thesis. Selecting the object oriented software modularity theory of Meyer (1997) as a starting point for measure development does not imply that this is the only possible choice or that the theory itself is entirely correct and comprehensive. What this choice does say is that Meyer's (1997) theory appears to offer a reasonable basis from which to develop a set of descriptive measures of object oriented software modularity. Only by developing, validating and using these measures can the original choice of theory be supported or disputed. If the theory should prove inadequate it can then be modified or discarded and the measures developed from it similarly treated. From a detailed understanding of object oriented software modularity (Meyer 1997), a set of measures can be developed to provide a detailed description of C++ class and object modularity.

2.4. Conclusion

It has been recognised that a formal process of software measure development should improve the ability of users to appropriately apply and interpret software descriptive measures (Briand, Morasca & Basili 2002, pp.1106-1107). The sets of software measures by Chidamber and Kemerer (1994), Abreu and Carapuca (1994), Bieman and Kang (1995), Briand, Morasca, and Basili (1999) and Arisholm, Briand and Foyen (2004) have been developed according to informal processes of measure development. To adapt these informal processes to the development of measures to provide a detailed description of a complex software characteristic, it is necessary to precisely specify how each stage of the measure development process should be executed and how the information produced at each stage should be presented. In this way, the design decisions of the measure developer are communicated to a potential user, allowing the user to determine whether or not the measures are appropriate to their proposed application.

This thesis describes a systematic process of descriptive measure development that specifies how each stage of measure development should be performed and how the products of these stages can be presented. Developing measures to provide a detailed description presents special challenges due to the large amount of information that must be communicated to a potential user. This process is demonstrated through the development of measures to provide a detailed description of C++ class and object modularity. Meyer's (1997, pp. 46-53) rules of modularity form the theoretical basis from which these measures are developed. This theoretical basis is presented as a sub-characterisation of modularity with associated conceptual definitions, and as a set of natural language models of C++ software that describe features of the software that

affect the levels of each modularity sub-characteristic present. Mathematical entity-relationship models of the software describe the software features identified in the natural language models. The descriptive software measures are unambiguously defined in terms of the mathematical model elements. The mathematical entity models and measures defined upon them are implemented within a software based measurement instrument. The outcome of the measure development process is a measurement instrument implementing a set of measures that describe the levels of modularity present in C++ class and object modules.

This thesis contributes to the field of software measurement by defining a process that supports the development of measures to provide a detailed description of a complex software characteristic. It also contributes to the field of software measurement by defining a set of measures that provide a detailed description of C++ class and object modularity. The systematic process of measure development illustrated in Figure 1-3 and described and demonstrated in Chapter 3 to 6 of this thesis facilitates the development of these measures of C++ class and object modularity. The Chapter 7 case study demonstrates validation of these measures with respect to a small software system, and presents the analysis and interpretation of the measurement data obtained by applying the measures to this software system.

The following Chapter 3 describes the conceptual definition stage of descriptive measure development. This stage is demonstrated by the sub-characterisation and conceptual definition of object oriented software modularity.

3. Conceptual Definition

This chapter describes the sub-characterisation and conceptual definition stage of software descriptive measure development. This stage corresponds to the shaded boxes in the Figure 3-1 diagrammatic representation of the measure development process. Section 3.1 of this chapter describes the conceptual definition stage in terms of its prerequisites, how it is performed and its products. Following this, section 3.2 demonstrates the steps of this stage by sub-characterising and conceptually defining object oriented class and object modularity based on the theory of Meyer's (1997, pp. 46-53) five rules of modularity.

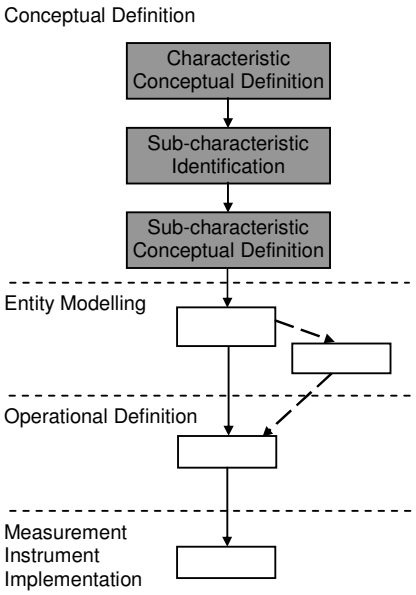


Figure 3-1 Conceptual definition stage of the measure development process

3.1. Stage 1 of measure development process - conceptual definition

In the conceptual definition stage, characteristic and sub-characteristic conceptual definitions are derived. "A conceptual definition defines a concept in terms of other concepts, the meaning of which is assumed to be more familiar to the reader." (Diamantopoulos & Schlegelmilch 1997, p.21) The conceptual definition is intended to "(a) capture the essence or key idea of the concept, and (b) distinguish it from other similar but, nevertheless, distinct concepts." (Diamantopoulos & Schlegelmilch 1997, pp.21-22). These conceptual definitions, together with the natural language model defined in the entity modelling stage, express the theoretical basis from which to develop descriptive software measures. It is important to ensure that the characteristics and sub-characteristics to be described by the measures are clearly identified and well defined since "a conceptual definition logically precedes an operational [measure] definition and, thus, it should be used to guide the development of the latter." (Diamantopoulos & Schlegelmilch 1997, p.22). The quality of the conceptual definitions will influence the

quality of the description of the software obtained from the measures subsequently developed based on these conceptual definitions. Conceptual definitions communicate the aspects of the software described by a set of measures and thus facilitate the analysis and interpretation of data obtained from applying these measures (Kitchenham 1996, p64).

It is uncommon to find software measures developed without some explicitly stated conceptual definition of the characteristic to be measured; however, Ferrett and Offutt (2002) provide one such example. They propose measures of modularity without first conceptually defining the particular aspects of modularity that each of these measures describe. This lack of explicit conceptual definition of modularity makes it difficult for a potential user of these measures to determine whether or not they are relevant to their own potential application. If the measures are applied, it then becomes difficult to interpret the measured data since the precise aspects of modularity described are not explicitly stated.

More usually, the characteristic to be described by the measures is defined in general terms that do not fully convey the theoretical basis from which the measures were developed. For example, Chidamber and Kemerer define object oriented class cohesion in general terms as "the internal consistency within parts of the design" (Chidamber & Kemerer 1994, p479). By contrast, their Lack of Cohesion in Methods (LCOM) measure (Chidamber & Kemerer 1994, p488) describes a particular aspect of cohesion related to the use of a common set of data by methods within a class. It would be easier for a potential user to apply and interpret this measure if the conceptual definition from which it was developed defined, in precise rather than general terms, the particular aspect of cohesion described.

The inability to precisely define a characteristic may indicate that it is not sufficiently well understood and that a better understanding is needed before descriptive measures can be developed. In the case of complex characteristics, it may not be possible to provide a single conceptual definition since the characteristic may be a cluster (Gasking 1960; Ellis 1966). Regarding such clusters, "The most important feature of a cluster is that it may be identified by any one of a large number of characteristics. But it has no definition in any commonly accepted sense of this word. For no one characteristic belongs essentially to a cluster. Our concept is formed by a continuing association of characteristics - an association which is relatively stable." (Ellis 1966, p34). When conceptually defining complex characteristics of software, such cluster characteristics must be initially defined in terms of the sub-characteristics that contribute to their manifestation in the software. The need to sub-characterise complex

software characteristics before measuring them has been recognised for many years. For example, Stevens, Myers and Constantine (1974) identify interface complexity, type of connection and type of communication as important sub-characteristics of software coupling. Fenton (1994, p199) also recognises sub-characterisation as a promising approach to the measurement of software complexity, stating that "…the most promising approach is to identify specific attributes of complexity and measure these separately". Meyer recognises that modularity is a complex characteristic of software, stating that "a single definition of modularity would be insufficient; as with software quality, we must look at modularity from more than one viewpoint." (Meyer 1997, p. 39).

Selection of the particular aspects of a characteristic to be measured should be made on the basis that they describe the characteristic in sufficient detail while at the same time excluding any unnecessary refinement to the description. The theoretical basis from which the measures are to be developed should include the identification of important sub-characteristics of the main characteristic to be measured. Difficulty in identifying a satisfactory set of sub-characteristics may indicate that the chosen theoretical basis is not sufficiently detailed. In this case, the theory may need to be further enhanced or perhaps discarded and a different theoretical basis used. Different theoretical bases may identify different sub-characteristics of the same major characteristic. Selection of the sub-characteristics to include and those to omit will also be guided by the degree of descriptive detail required from the measures developed. The greater the degree of descriptive detail inherent in the sub-characterisation, the more detailed the description that can be obtained from measures developed from this sub-characterisation.

Software engineering is a relatively new area of investigation and measure development may be hindered by a lack of theoretical understanding of characteristics for which it is desired to develop descriptive measures. Using as a basis for measure development a tentative or incomplete theory may result in measures being developed to describe aspects of a software characteristic that are unimportant or unrelated or may result in important aspects of a characteristics being overlooked, with no measured defined to describe them. The only way to overcome the problem of lack of theoretical understanding of software development is to perform research. One area of research could be the definition, validation and application of measures developed according to the current theoretical understanding of software. Through this, greater understanding of software could be achieved, further enhancing the available theory and allowing for the definition of better measures. The systematic process of measure

development described in this thesis supports the development of descriptive software measures regardless of the sophistication of their theoretical basis. This theoretical basis is described in the conceptual definition and natural language models of the software.

The following sections 3.1.1, 3.1.2 and 3.1.3 describe the conceptual definition stage of measure development in terms of its prerequisites, performance and products.

3.1.1 Prerequisites to the conceptual definition stage

The theory elements required as inputs to the systematic descriptive measure development process are the conceptual definition of the characteristic of interest and the identification of software features that affect the levels of this characteristic present in the software. A software expert's intuitive understanding of the characteristic, and an established theory regarding the manifestation of the characteristic within the software are both examples of theoretical bases from which descriptive measures can be developed. Where the characteristic to be described by the measures is a complex concept, the theoretical basis for measure development may need to identify and define some important sub-characteristics of the characteristic to be measured.

Since measures are developed based on the expression of this theory, the quality of this theory will affect the quality of the measured description obtained. An incomplete theory will produce a conceptual definition and natural language model of a software characteristic that is similarly incomplete. Models are intended to capture only the relatively important features of an entity while omitting less important features. They will always provide an incomplete description and this is desirable, as irrelevant details will confuse the person analysing the modelled information. Thus, it is not the incompleteness of the theory used to form the basis of measure definition that is a problem, but rather the possibility that important features have been omitted from the model and unimportant features included. A similar problem can arise when a speculative theory is used to derive the characteristic conceptual definitions and natural language model because again there is the possibility that important features have been omitted from the model and unimportant ones included. The systematic measure development process is sensitive to these problems because they may lead to measures being developed for unimportant features and none being developed for important ones.

Measure validation provides a means to evaluate the sufficiency or otherwise of the theoretical basis from which a set of measures was developed. Within the measure development process,

provision is made for a content type validation. This is a subjective assessment of the "extent to which a measure appears to measure the characteristic it is supposed to measure." (Diamantopoulos and Schlegelmilch 1997, p.34). This type of validation is included in the measure development process because all the information needed to perform it can be found in the products of the process itself. Once a set of measures have been defined to describe a software characteristic, and have been shown to have a sufficient level of content validity for a proposed application, a construct type validation can be performed to determine the extent to which the measures “relate to other measures consistent with theoretically derived hypotheses concerning the concepts (or constructs) that are being measured.” (Carmines and Zeller 1979, p. 23) For example, if software theory suggested that software systems comprised of modules with low modularity were more difficult to maintain, then modularity and maintainability measurement data collected from a software system could be analysed to determine whether or not this relationship was upheld. If it was upheld then this could be used as evidence of the validity of the modularity measures. If it was not upheld, then it could be used as evidence that the modularity measures provided a poor description of the software. As it could also be used as evidence of poor measures of maintainability or poor theory regarding the relationship between modularity and maintainability, this type of validation is more about accumulating evidence of validity rather than proving it conclusively.

At the current state of software development understanding, the question may arise as to whether the prevailing theoretical understanding of a software characteristic is sufficient to form the basis of descriptive measure development. To answer this question, a measure developer should gather as much information regarding the characteristic of interest as possible and then make a subjective assessment as to whether this information provides a sufficiently detailed description of the characteristic. If it has sufficient detail, then measure development should proceed. If it has insufficient detail, then measures should not be defined based on this theory. Any measures defined to describe a software characteristic should be validated before they are widely accepted.

3.1.2 Performance of the conceptual definition stage

Once the prerequisites have been met, the conceptual definition of the characteristic to be measured can take place. Figure 3-2 describes the process of conceptual definition. The first step is to identify and state the characteristic that will be described by the measures. If this is a simple characteristic, then it can be immediately conceptually defined. If it is a cluster

characteristic then the major sub-characteristics that identify it will need to be stated and each of these conceptually defined. As with any development process, it may take several sub-characterisation and conceptual definition iterations to achieve a satisfactory result.

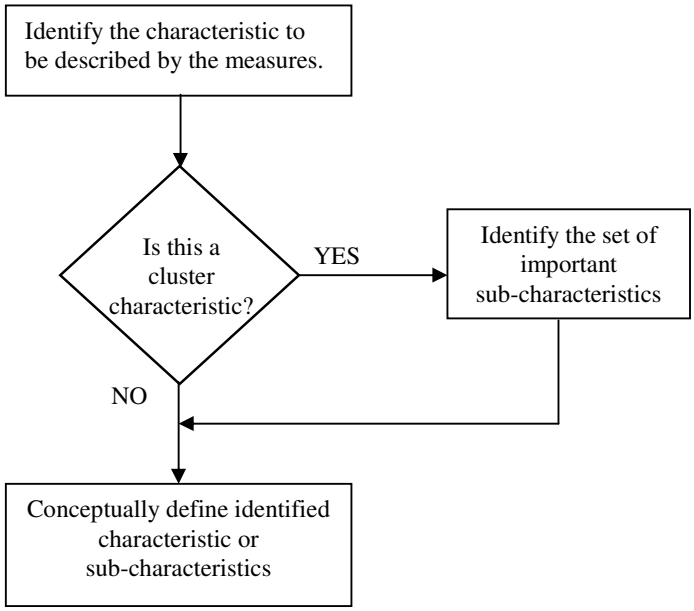


Figure 3-2 Process of conceptual definition of characteristic to be described by measures

3.1.3 Products of the conceptual definition stage

The products of the conceptual definition stage will vary depending on the complexity of the characteristic to be described by the measures. For a simple characteristic that most people understand well, the conceptual definition stage will result in a single dictionary like definition of the characteristic. For a more complex cluster characteristic that is not as well understood or does not have a single, widely accepted definition, it will be necessary to identify its most important sub-characteristics and conceptually define these. The product of the conceptual definition stage for a cluster characteristic is a sub-characterisation with an associated set of conceptual definitions for each of the most refined sub-characteristics.

3.1.4 Practical Considerations

Of all the stages of the systematic process of descriptive measure development, the conceptual definition stage is the most subjective in that the quality of the final product is dependent on the skill of the measure developer. This is particularly the case when descriptive measures of a

cluster type characteristic are developed. The measure developer is responsible for identifying those sub-characteristics that are the most important to measure. They are guided in this decision by their understanding of theoretical basis selected for measure development, the ultimate use to which the measures will be put and their own levels of experience and intuitive understanding of the characteristic to be measured. The measure developer is also responsible for recognising at what stage the sub-characterisation has resulted in sufficiently simple conceptual definitions. What may be a simple concept to one person may be beyond the comprehension of a less expert person. These issues can make the conceptual definition stage difficult to execute.

The systematic process of measure development advocated in this thesis is not intended to change the subjective nature of sub-characterisation and conceptual definition, and may not even make it easier. What it is intended to do is to communicate the decisions made by the measure developer to potential users of the measures. The sub-characterisation and conceptual definitions describe the particular aspects of the characteristic chosen to be included in the measured description. A future user is able to examine these and decide whether or not the measures will be sufficient for their purposes. If the sub-characterisation is too detailed, then they can choose to omit some measures. If it lacks detail, then they can choose to abandon their use of the measures, or further refine the sub-characterisation and develop new measures that provide a more detailed description of the software.

Performing a specific conceptual definition stage is important because it enables a measure developer to communicate the precise aspects of a characteristic that will be described by the set of measures subsequently developed. Without explicitly stated conceptual definitions, a potential user of software measures must rely on their own intuitive understanding of the described characteristic in order to validate the measures and later interpret the measured data. If this understanding is different from that of the person who developed the measures, then the measures may be incorrectly applied and interpreted. An explicit statement of sub-characterisation and conceptual definitions enables a user to understand exactly what the measures describe and from this understanding, correctly apply and interpret the set of descriptive measures. Together with the natural language entity model, the conceptual definitions describe the theoretical basis from which a set of descriptive measures is developed.

The remainder of this chapter demonstrates the conceptual definition stage for the development of descriptive measures of C++ class and object modularity.

3.2. Conceptual definition of C++ class and object modularity

The conceptual definition of C++ class and object modularity is the first stage of the development of measures to describe this software characteristic. All subsequent stages of the measure development process are dependent on these definitions. In conceptually defining C++ class and object modularity, the final measurement instrument implementation stage is considered in that, only measures that can be successfully implemented in the intended manner will be developed. The measures of C++ class and object modularity that will be developed and implemented here and in subsequent chapters of this thesis are intended to be implemented in a software based measurement instrument that analyses the structure of C++ code and from this extracts the measurement data. This means that only measures describing aspects of software structure can be implemented and hence, in this thesis only sub-characteristics of modularity related to software structure are conceptually defined.

3.2.1 Prerequisites

The theoretical basis of measure development selected as the prerequisite to conceptual definition is Meyer's (1997, pp.46-53) five rules of object oriented software modularity. From these rules of modularity, four major modularity sub-characteristics related to software structure are identified, further sub-characterised and conceptually defined. This results in a set of modularity sub-characteristics for which descriptive measures can be defined. A potential user of the measures developed based on this theory, as expressed in the modularity conceptual definitions, should satisfy themselves that the aspects of modularity they are interested in describing are the same as those covered by the conceptual definitions.

3.2.2 Sub-characterisation of object-oriented software modularity

Modularity is considered a cluster type characteristic (Gasking 1960; Ellis 1966) for which a single definition is insufficient (Meyer 1997, p.39) and so it must be sub-characterised before it is conceptually defined. Meyer (1997) defines five rules of modularity. These five rules form the basis of the modularity sub-characterisation and conceptual definition described in this chapter. This is fundamental to the modularity descriptive measure development process as, in combination with the natural language model, the modularity conceptual definition describes the theoretical basis from which the measures are derived.

Meyers (1997 pp. 46-53) five rules of modularity state that a module with a high level of modularity exhibits:

1. Direct mapping
2. Few interfaces
3. Small interfaces
4. Explicit interfaces
5. Information hiding

Before discussing Meyer's (1997) descriptions of these modularity sub-characteristics, it is important to note that, except for very simple characteristics, no sub-characterisation is going to include all possible sub-characteristics that define a characteristic. The most important point in identifying relevant sub-characteristics is to choose those that provide an adequate description of a characteristic. The distinction between define and describe is an important one. It takes all possible sub-characteristics associated with a particular characteristic to completely define it, but only a selected sub-set of these to describe it (Ellis 1966). Meyer (1997) has selected the direct mapping, few interfaces, small interfaces, explicit interfaces and information hiding sub-characteristics of modularity to describe object oriented software modularity. In a similar manner, Meyer (1997), in describing each of these modularity sub-characteristics, has identified certain software features that he believes affect each sub-characteristic. Again, these may not represent all the possible features affecting the sub-characteristic however they are the ones Meyer (1997) believes to have a significant effect and so, these are the ones that measures, developed according to Meyer's (1997) modularity theory, should quantify.

Meyer's rules of modularity can be divided into three distinct categories. The direct mapping rule concerns the relationship between the software structure and the structure of the problem domain. The information hiding rule concerns the internal structure of an individual module. The few interfaces, explicit interfaces and small interfaces rules concern the connections from individual modules to software elements external to themselves. In general terms, these last three rules describe module external coupling.

The following sections examine each of Meyer's modularity rules in detail.

3.2.2.1 Direct mapping

Meyer's rule of Direct mapping states that:

> "The modular structure devised in the process of building a software system should remain compatible with any modular structure devised in the process of modelling the problem domain" (Meyer 1997, p. 47)

To describe the extent to which a software system is structured in accordance with the rule of direct mapping, both the modular structure of the problem domain and of the software system itself needs to be measured and the two compared. In this thesis, the decision has been made to exclude direct mapping from the measured description of class and object modularity to be developed. There are two main reasons for this exclusion. Firstly, the modular structure of the problem domain would be most likely to map to a high level view of the software with modules comprised of several classes and objects. The aim in the thesis is to derive measures of modularity for individual classes and objects rather than groups of them. Thus the direct mapping rule is not directly applicable to the description of modularity to be developed in this thesis. Secondly, to determine the extent to which a module complies with the direct mapping rule, it is necessary to measure the modular structure of the problem domain document and the software document. The aim in this thesis is to develop measures only for the description of modularity that can be obtained from the software document alone. The implication of excluding direct mapping from the modularity description obtained from the measures developed in this thesis is that this description is incomplete according to Meyer's (1997) definition of modularity. A potential user of the measures should decide whether or not the incomplete description is sufficient for their needs. Depending on the particular measurement situation, the analysis and interpretation of the measured data may need to take into account the fact that direct mapping is not described by the measures.

3.2.2.2 Information Hiding

Meyer's rule of Information Hiding states that:

> "The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules" (Meyer 1997, p. 51)

The rule of information hiding is important to the measurement of class and object modularity as it is the only one of Meyer's rules that can be applied to modules as individuals rather than to groups of connected modules. According to Meyer (1997, p 51-53), a module with a high level of information hiding, and thus high modularity, exhibits the following.

1. That "every module is known to the rest of the world … through some official description, or **public** properties" … "The public properties of a module are also known as the **interface** of the module" (Meyer 1997, p. 51).
2. This "description should only include *some* of the module's properties. The rest should remain non-public, or **secret**. (Meyer 1997, p. 51).
3. "We may picture a module supporting Information Hiding as an iceberg; only the tip – the interface – is visible to the clients." (Meyer 1997, p. 51).
4. "As a general guideline, the public part should include the specification of the module's functionality; anything that relates to the implementation of that functionality should be kept secret, so as to preserve other modules from later reversals of implementation decisions" (Meyer 1997, p. 52)
5. "Assume a module changes, but the changes apply only to its secret elements, leaving the public ones untouched; then other modules who use it, called its *clients*, will not be affected. The smaller the public part, the higher the chances that changes to the module will indeed be in the secret part" (Meyer 1997, p51).
6. "Information hiding emphasizes the separation of function from implementation." (Meyer 1997, p. 52).
7. "the use of abstract data types as the source of our modules gives us a practical, unambiguous guideline for applying information hiding in our designs" (Meyer 1997, p145)

Points 1, 2 and 3 refer to the general structure of the module and the size of the public module interface relative to the hidden part of the module. Points 4, 6 and 7 refer to what should be included in each of the interface and hidden parts of the module. Point 5 refers to the size of the module interface and also to the requirement that changes to the module should not affect the operation of the interface.

At this stage, it is helpful to interpret these points to obtain a representation of a module with a high level of information hiding. Figure 3-3 shows such an object oriented module.

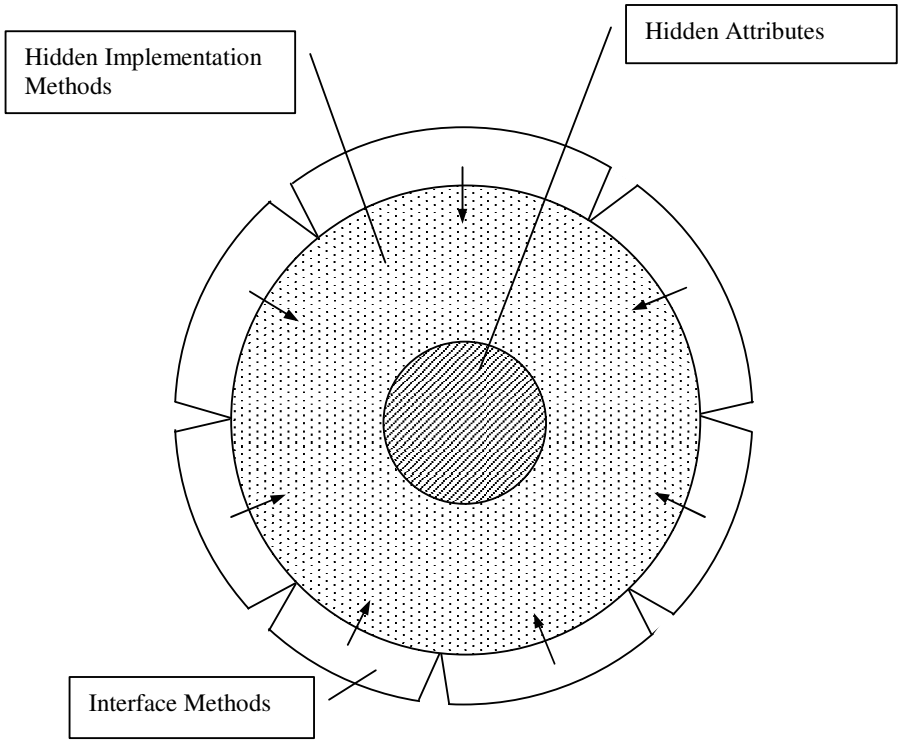


Figure 3-3 Representation of an object oriented module with a high level of information hiding

In accordance with points 1, 2 and 3, the module consists of public interface and private hidden parts where the interface is smaller than the hidden part. In accordance with points 4, 6 and 7, the interface only contains methods while the hidden part contains data attributes and implementation specific methods. Point 5 requires that the interface of a module be as stable as

possible. Ideally, changes to the implementation of the module do not affect the operation of the interface. In accordance with this, Figure 3-3 shows that interface methods do not directly access the module data. Should it be necessary to alter the operation of the interface, then ideally, changes are restricted to as few interface elements as possible. To this end, the arrows on Figure 3-3 indicate that the separate interface methods only invoke hidden methods. This means that changes to an interface method do not flow on to affect any other interface methods that may directly, or indirectly via hidden methods, invoke the changed interface method.

To measure the degree of information hiding exhibited by a module, it is necessary to identify the characteristics of the module that provide an adequate description of information hiding. This is a subjective judgement on the part of the measure developer. Ideally, sufficient characteristics are selected to provide a sufficiently detailed description without introducing unnecessary information.

Looking a the points describing Meyer's view of information hiding, it can be seen that points 1, 2, 3, 4, 5, 6 and 7 are all concerned with the independence of the interface from the implementation details of the module. This can be characterised as the interface independence of a module. The greater the interface independence of a module, the greater its level of information hiding and hence, the greater its modularity. To simplify the analysis of the measurement data, the decision has been made to define measures such that a value of zero indicates high modularity and increasing values indicate decreasing modularity. To have this property, it is the dependence rather than independence of the module interface that will be described by the measures.

From the previous discussion, it can be inferred that there are two types of interface dependence. The first is **interface implementation dependence** which describes the dependence of the interface on implementation specific elements of the module. This can be further sub-characterised as the **size of the interface** (Points 1, 2, 3 and 5) and as the presence of implementation specific data elements exposed in the interface or **data exposure** (Points 4, 6 and 7). The second type of interface dependence is due to the **interface element interdependence** (Point 5). The sub-characteristics written in bold, in accordance with Meyer's description of module information hiding, together describe the levels of interface dependence of an object oriented module. At this stage of measure development, validation of this sub-characterisation is by subjective assessment of the sub-characterisation against the theory from which it was derived.

3.2.2.3 Few Interfaces

Meyer's rule of few interfaces states that:

"The few interfaces rule restricts the overall number of communication channels between modules in a software architecture: Every module should communicate with as few others as possible." (Meyer 1997, p. 47)

According to Meyer (1997, p 47- 48), a module with few interfaces, and thus high modularity, exhibits the following.

1. "The few interfaces rule restricts the overall number of communication channels between modules in a software architecture: Every module should communicate with as few others as possible."
2. "Communication may occur between modules in a variety of ways. Modules may call each other (if they are procedures), share data structures etc. The Few Interfaces rule limits the number of such connections." (Meyer 1997, p. 47)
3. "… if a system is composed of *n* modules, then the number of intermodule connections should remain much closer to the minimum, *n-1* … than to the maximum *n(n-1)*." (Meyer 1997, p. 47)

Point 1 identifies the communication channels between modules as establishing intermodule interfaces. Point 2 provides an open ended example of types of communication between modules that establishes these interfaces and further describes them as connections. These connections can be direct, such as procedure calls, or indirect, such as shared data structures. The few interfaces rule is not concerned with the degree of connection between modules established by a communication channel, but rather the existence of a communication channel or connection between modules. The explicit interfaces (Meyer 1997, p50) and small interfaces (Meyer 1997, pp. 48-50) rules describe the degree of connection between modules whereas the few interfaces rule is concerned only with the existence of the communication channel. Point 3 states that a module with high modularity has few connections with other modules.

In developing measures of modularity in the thesis, the few interfaces rule will be interpreted as meaning that a module with high modularity has few connections or relationships with other modules along which communication can occur. This describes the potential for

communication rather than the degree communication between modules. It provides a high level description of module connections. Adherence to the few interfaces rule will be described by measures quantifying the number of **external relationships** a module has. The fewer the number of such external relationships, the greater the modularity of a module. The term relationship has been selected rather than the term communication used by Meyer (Meyer 1997, p. 47) because communication implies some form of information exchange where as the term relationship only implies some sort of connection.

Extending the graphical representation of a module with high modularity, Figure 3-4 represents external relationships from module A to module B. These relationships are like roads between the modules. The external relationship sub-characteristic of modularity describes only the existence of such roads and the general form they take. It does not describe whether or not these roads are used, how difficult it is to travel along them or where their start or end points are.

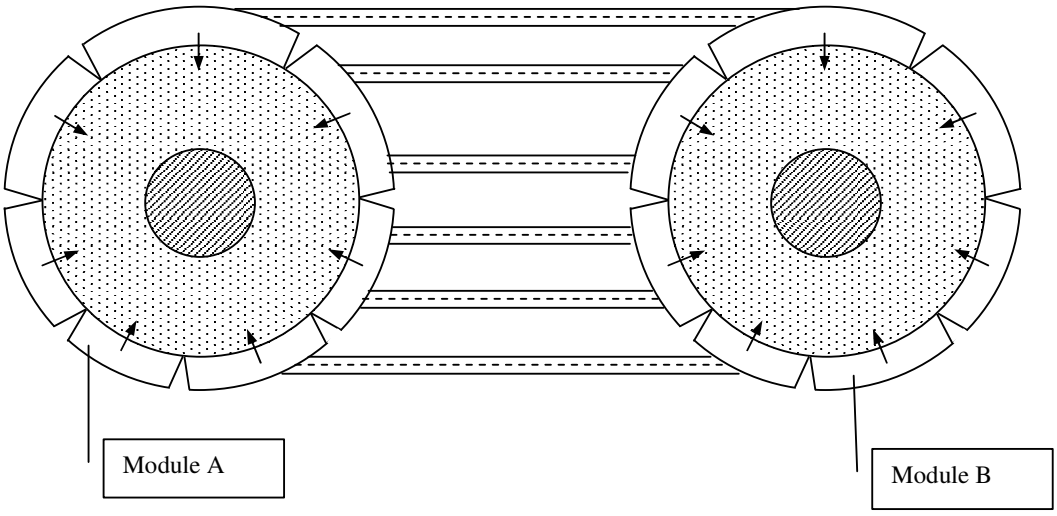


Figure 3-4 Representation of object oriented module external relationships

3.2.2.4 Explicit Interfaces

Meyer's rule of explicit interfaces states that:

"Whenever two modules A and B communicate, this must be obvious from the text of A or B or both." (Meyer 1997, p. 50)

According to Meyer (1997, p 50), a module with few interfaces, and thus high modularity, exhibits the following.

1. "Whenever two modules A and B communicate, this must be obvious from the text of A or B or both." (Meyer 1997, p. 50)
2. "… if you need to decompose a module into several sub modules or compose it with other modules, any outside connection should be clearly visible" (Meyer 1997, p. 50)
3. "… it should be easy to find out what elements a potential change may affect" (Meyer 1997, p. 50)
4. "… how can you understand A by itself if B can influence its behaviour in some devious way?" (Meyer 1997, p. 50)
5. "… there is more to intermodule coupling than procedure call; data sharing, in particular, is a source of indirect coupling." (Meyer 1997, p. 50)

In the previous discussion of the few interfaces rule, interfaces between modules were determined to exist when relationships were established that created a communication channel between the modules. The explicit interfaces rule further refines on the information provided by the few interfaces rule by describing the type of interface through which these connections should take place. From Point 1, the explicit interfaces rule requires that the existence of these relationships be able to be discerned from the text of the modules. The measures in this thesis are intended to describe aspects of modularity that can be automatically measured from C++ source code. This means that the text referred to in the definition of the explicit interfaces rule is interpreted to mean the C++ source code that declares and implements the class and object modules. Any accompanying comments text will not be measured. Points 2, 3, 4 and 5 refer to the type of connections between modules that affect modularity. Outside connections should be clearly visible (Point 2), easy to discern (Point 3), not devious (Point 4) and not indirect (Point 5). For a high level of modularity, the connections between modules should have a minimum level of obscurity.

From Points 1 and 2, modularity is reduced when the connections between modules are due to **unstated relationship**s. From Point 4, modularity is reduced when the connections between modules are devious or **non-standard connections**. Devious connections may be ones that are due to an **unexpected relationship** between modules, or may take place **via non-standard interfaces**. From Point 5, indirect or **distant connections** between modules will increase connection obscurity. From Point 3, the connection between modules should be stable, and not vary thus modularity is maximised when the number of **variable connections** are minimised.

Extending the graphical representation of module external relationships, Figure 3-5 represents connection obscurity from module A to module B. Connection obscurity describes the type of roads between modules. They may be straight or curved, direct or indirect via a software element such as a shared global variable. They may start or end at any point within the modules and it may be difficult to see where they go at all. The connection obscurity sub-characteristic of modularity does not describe whether or not these roads are used.

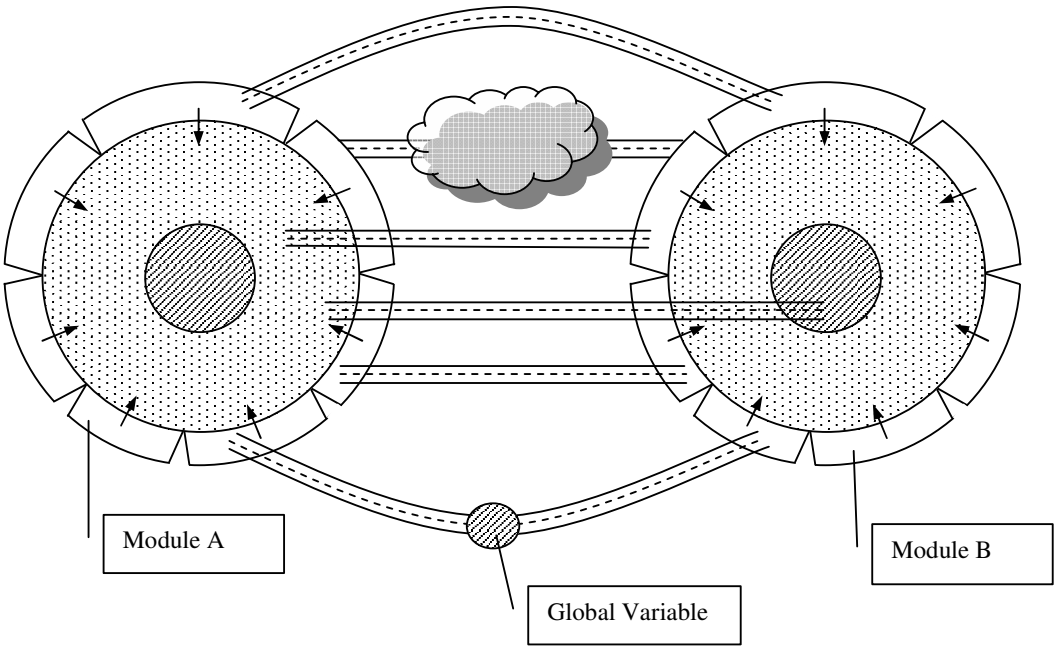


Figure 3-5 Representation of object oriented module connection obscurity

3.2.2.5 Small Interfaces

According to Meyer's rule of small interfaces

1. "If two modules communicate, they should exchange as little information as possible." (Meyer 1997, p. 48)
2. "The Small Interfaces or "Weak Coupling" rule relates to the size of intermodule connections rather than to their number." (Meyer 1997, p. 48)
3. "… the channels of communication between modules must be of limited bandwidth." (Meyer 1997, p. 48)
4. Shared data between modules is identified as a way in which the size of the interface between modules is increased. (Meyer 1997, pp 48-49)

Modularity is maximised when the size of a module's interface is minimised by restricting the amount of information flowing across the interface. As discussed in the Few Interfaces rule, communication between modules takes place via communication channels. As Point 1 above states, the small interfaces rule is not concerned with the capacity of these communication channels but rather with the amount of information passed along them. Modules communicate by sharing data and services with each other. The modularity of a module is reduced when it depends on another module for some of the services or data it needs to perform its own tasks correctly. Thus, one way that interface size can be characterised is as the degree to which a module is dependent on other modules for services or data. In object oriented software, the operation of a module may depend on **services invoked** from other modules or on **interface elements inherited** from other modules. The state of a module, stored in the data elements it reads, may also depend on **external variables read** by the module or on **external functions writing** values to these data elements.

Extending the graphical representation of module connection obscurity, Figure 3-6 represents dependency of module A on module B. in simple terms, dependency describes the extent to which a module makes use of and is dependent on the road connections to other modules in order that it can correctly accomplish its own operations. A module with high modularity is a complete unit, able to perform its required tasks with no reference to external software elements. In practice, such a module does not exist as modules need to connect to other modules to form a functioning software system. These connections can create dependencies between modules. Not all connections increase a module's dependency. For example, a module may write a value to a global variable but never read from it. The correct functioning of the

module is not dependent on the value stored in the global variable. Another module may read from this same global variable but never write to it. This second module is dependent on the first module for the correct maintenance of the global variable. The second module is thus dependent on the first. It may also be the case that a connection exists between modules that has the potential to reduce the modularity of one or both modules however doesn't because neither module uses this connection. For example, in Figure 3-6, a communication channel is shown from the hidden methods of Module A to the hidden attributes of Module B. The diagram indicates that neither module uses this channel and so, it does not produce a dependency between them.

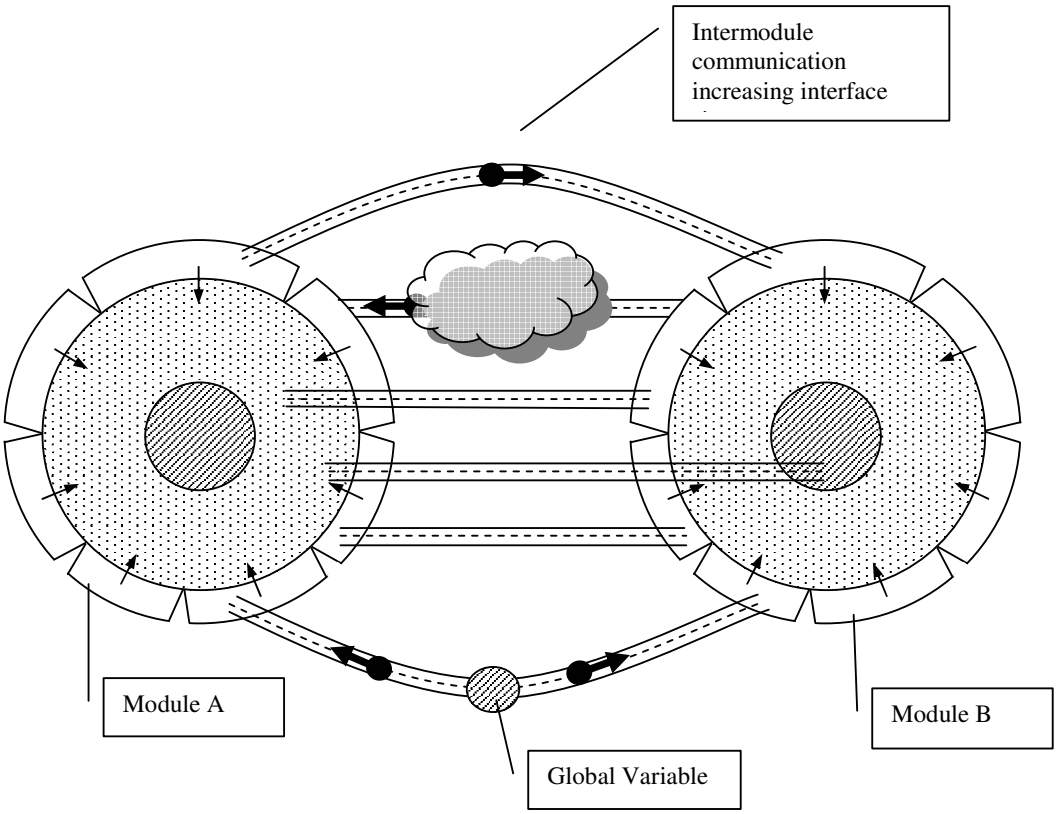


Figure 3-6 Representation of object oriented module dependency

Figures 3-3, 3-4, 3-5 and 3-6 are simplified representations of object oriented modularity that describe rather than define the modularity of class and object modules. Interpreting Meyer's (1997) rules of modularity leads to the identification of modularity sub-characteristics that together provide a description of the levels of modularity present in an object oriented class or

object module. Figure 3-7 summarises the characterisation and sub-characterisation of modularity according to Meyer's five rules (Meyer 1997, pp. 46-53). Following this, these characteristics and sub-characteristics are conceptually defined.

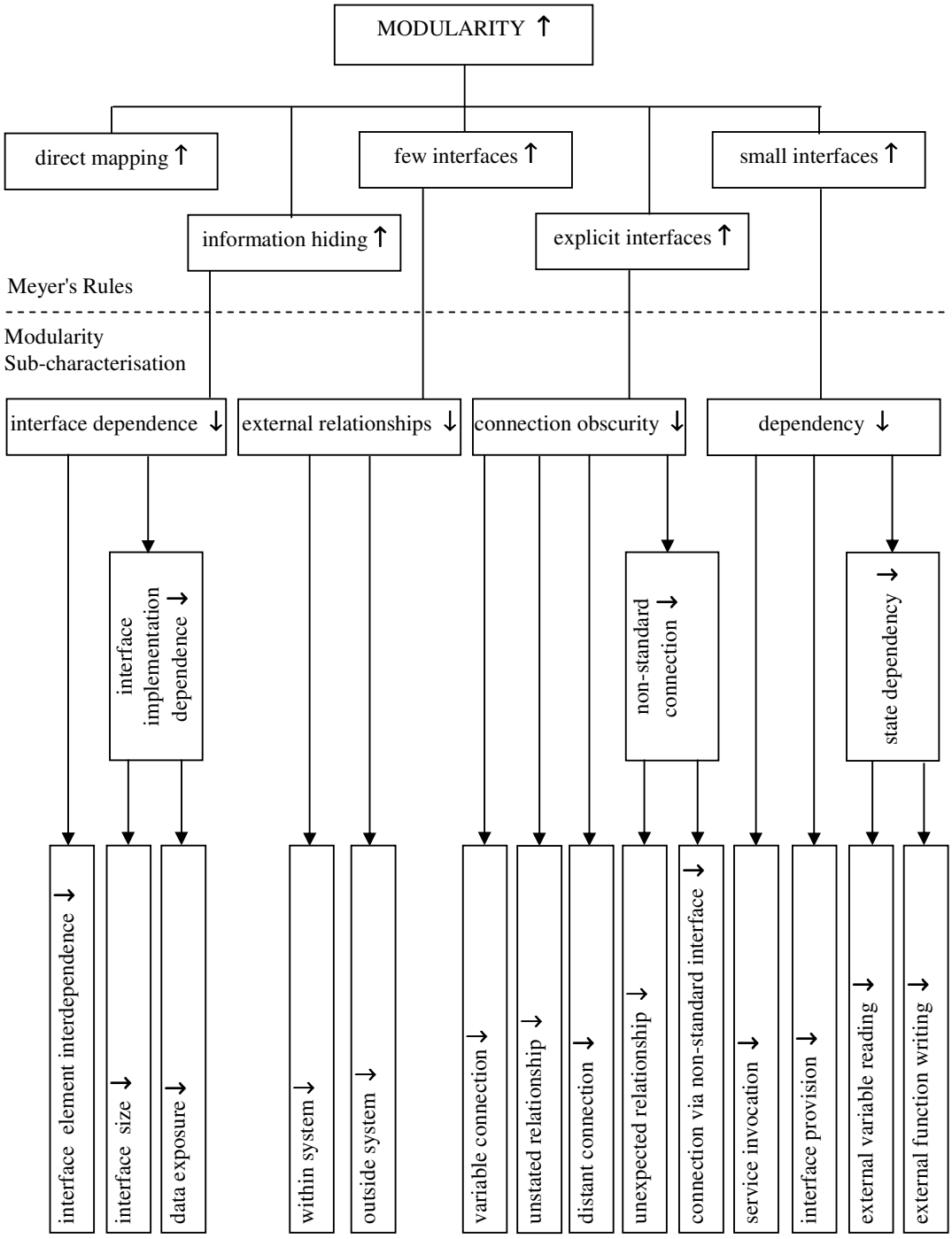


Figure 3-7 Modularity sub-characterisation based on Meyer's (1997) rules of modularity

Figure 3-7 illustrates Meyer's (1997) rules of modularity and the associated modularity sub-characteristics selected to describe them. The extent to which a module complies with the

- information hiding rule is described by the interface dependence sub-characteristic of modularity.
- few interfaces rule is described by the external relationships sub-characteristic of modularity.
- explicit interfaces rule is described by the connection obscurity sub-characteristic of modularity.
- small interfaces rule is described by the dependency sub-characteristic of modularity.

The direct mapping rule is not included in the sub-characterisation and so will not be described by the measures developed in this thesis. The reason for this omission is that, while the information required to describe compliance with the few interfaces, small interfaces, explicit interfaces and information hiding rules can be collected from the software system alone, evaluating direct mapping requires information from the software system and the problem domain. The measures developed within this thesis are intended to be collected from the software system source code alone and so, compliance with the direct mapping rule cannot be measured. Omitting the direct mapping rule from the sub-characterisation reduces the detail of the description that is obtained from measures developed from it. This reduction of detail should be considered before the measures are applied to a task. If it is important to describe direct mapping then new measures will need to be defined to supplement the existing set.

The arrows in Figure 3-7 indicate that increasing (↑) levels of direct mapping, few interfaces, small interfaces, explicit interfaces and information hiding increase modularity while decreasing (↓) levels of the selected modularity sub-characteristics result in increased modularity. For example, to increase modularity, Meyer advocates that a software system have increasing levels of modules with few interfaces (Meyer 1997, p. 47), indicated by the up arrow (↑) in the few interfaces box in Figure 3-7. To achieve this increase in modules with few interfaces, individual modules should have minimal levels of external relationships, indicated by a down arrow (↓) in the external relationships box in Figure 3-7. Interface dependence, external relationships, connection obscurity and dependency sub-characteristics were specifically selected to describe modularity because they have the property that a value of zero indicates maximum modularity and values above zero indicate decreasing modularity. This property will ultimately simplify the data analysis and interpretation phases of measurement.

The lowest level of sub-characteristics in Figure 3-7 represents the most refined level of sub-characterisation. These sub-characteristics are conceptually defined and measures will be developed to directly describe them. Descriptions of the intermediate sub-characteristics of state dependency, non-standard connection and interface implementation dependence are derived from the analysis of their associated immediate sub-characteristics. Description of the highest level sub-characteristic of external relationships, dependency, connection obscurity and interface dependence sub-characteristics must be similarly derived from their associated immediate low level and intermediate level sub-characteristics.

The following section describes the conceptual definitions associated with the modularity sub-characterisation illustrated in Figure 3-7.

3.2.3 Conceptual definition of modularity sub-characteristics

An important point to note before discussing the conceptual definition of modularity is that high levels of modularity sub-characteristics do not necessarily equate to high levels of software quality. For example, a module with no interface elements at all is considered to have a very high level of the information hiding sub-characteristic of modularity since all of its elements are hidden. Such high modularity does not equate to high quality because such a module cannot normally be accessed by other elements within a software system and so is not very useful. When reading the conceptual definitions of the modularity sub-characteristics, it is important to bear in mind that they are only intended to describe modularity, and not other aspects of software quality.

As a cluster type characteristic (Gasking 1960; Ellis 1966), object oriented software modularity is conceptually defined by sub-characteristics.

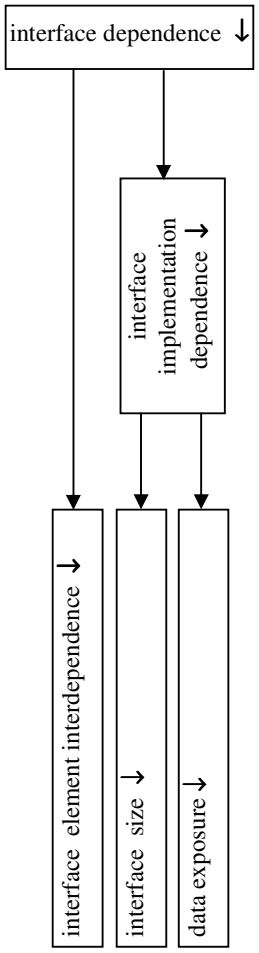
Modularity -

> *The degree to which a module exhibits direct mapping, few interfaces, small interfaces, explicit interfaces and information hiding. (Meyer 1997, p.46)*

The following conceptual definitions define the modularity sub-characteristics illustrated in Figure 3-7 in relatively simple terms that a potential user can understand. These conceptual definitions are derived from Meyer's (1997, pp.46-53) explanation of his rules of object oriented software modularity.

3.2.3.1 Interface dependence sub-characteristic

The interface dependence sub-characteristic of modularity describes the degree to which an object oriented class or object module complies with the rule of information hiding.



Interface Dependence

The degree to which the module interface elements are dependent on each other and on the implementation specific details of the module.

Interface dependence is sub-characterised as interface element interdependence and interface implementation dependence.

- Interface Element Interdependence

  The dependency producing connections between interface elements within the same module.
- Interface Implementation Dependence

  The dependence of the interface elements of a module on the implementation specific elements of the same module.

  Interface implementation dependence is further sub-characterised as interface size and data exposure.
- Interface size

  The degree to which implementation specific code is included in the module's interface.
- Data exposure -

  The degree to which the module data is revealed to the module interface.

Figure 3-8 Interface dependence sub-characterisation and conceptual definition

Meyer's (1997, pp.51-53) rule of information hiding dictates that to optimise information hiding, a module should have a small, implementation independent interface through which a client may access the services of the module. These services should be implemented within the larger, hidden, implementation dependent part of the module. A module is likened to an iceberg where the tip is the implementation independent interface and the majority of the iceberg,

containing the implementation dependent details, is hidden below the water (Meyer 1997, p.51). A primary rationale of information hiding is that changes to the implementation of a module should not affect the operation of the module's interface since changes here may impact on the clients of the module (Meyer 1997, p.51). This can be supported by ensuring that the module interface elements operate as independently as possible from the implementation details of the module, and from each other. This last requirement is needed to ensure that should a change to an interface element be necessary, this change does not flow on to other interface elements that directly or indirectly depend on it. From this view of information hiding, the following interface dependence sub-characteristics are derived.

Interface element interdependence refers to the connections between the interface elements of the module. If one interface element accesses another, then a change to the accessed element could affect the accessing element and hence have a greater impact on the module's interface and possibly on clients that use it. If the interface elements are not directly or indirectly dependent on each other, then any changes made directly to one interface element will not cause changes to propagate to any other interface elements that depend on it.

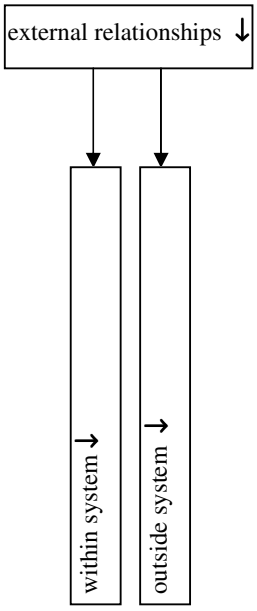
Interface implementation dependence describes the extent to which the interface elements of a module are dependent on the implementation specific details of the module. Where the interface elements are strongly dependent on the implementation details, changes to these details may cause more than minimal changes to the interface elements that rely on them. Interface implementation dependence is further sub-characterised as interface size and data exposure.

Interface size describes an aspect of interface implementation dependence in that the larger the interface, the greater the chance that it contains implementation dependent details. This idea comes from the previously described analogy of a module as an iceberg (Meyer 1997, p.51).

Data exposure is another way that the module interface can be dependent on the implementation details. Data elements are considered to be highly implementation specific. As such, they should be contained within the hidden section of the module rather that appearing in the interface (Meyer 1997, p.18). Data exposure is also increased when the interface elements directly access data elements. The implementation dependence of the interface is increased when module data is revealed to the interface.

3.2.3.2 External relationships sub-characteristic

The external relationships sub-characteristic of modularity describes the degree to which an object oriented class or object module complies with the rule of few interfaces.



External Relationships

The degree to which a module has relationships with elements external to itself.

External Relationships is sub-characterised as relationships within the system and relationships outside the system.

- Within the system

  The relationships between a module and elements external to the module but still within the given software system.
- Outside the system

  The relationships between a module and elements external to the module and outside the given software system.

Figure 3-9 External relationships sub-characterisation and conceptual definition

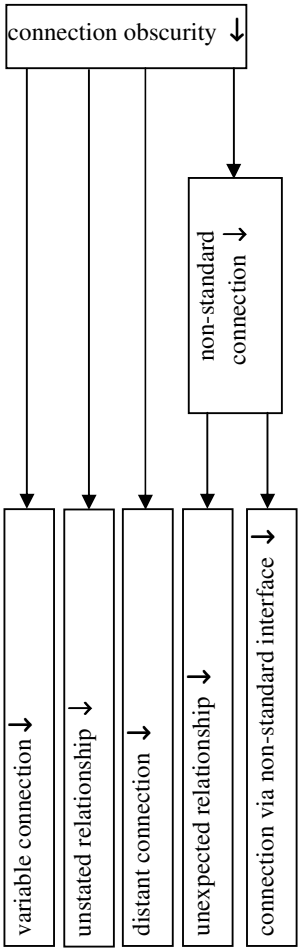
The few interfaces rule dictates that a module with optimal modularity communicates with as few external elements as possible (Meyer 1997, p.47). Meyer uses the term *communicate* to mean a type of connection between modules that establishes a relationship between them. This need not be a direct connection since he gives an example of two modules related to each other via a shared data structure (Meyer 1997, p.47). In this thesis, adherence to the few interfaces rule will be determined by describing the *relationships* between a module and elements external to that module. The term *relationship* has been selected rather than the term *communication* because communication implies some form of information exchange where as the term relationship only implies some sort of connection. From this view of the few interfaces rule, the following external relationships sub-characteristics are derived.

External relationships within the system occur between a module and an external element that is within the measured software system. The relationship can be determined from examination of the module and any system elements that participate in the relationship. For example, the ancestor classes of a child class can be determined when all these classes are within the measured software system.

External relationships outside the system occur between a module and an external element that is outside the measured software system. Often the full relationship cannot be determined from examination of the module and any system elements that participate in the relationship. For example, the full set of ancestor classes of a child class cannot be determined when one or more of these ancestor classes lie outside the measured software system. The measures developed in this thesis are intended to be collected from the source code within an identified software system. The relationships between a system module and elements outside the system cannot be completely measured. For this reason, measures will not be developed to describe external relationships outside the system sub-characteristic of modularity. A user of the measures should ensure that the measured software system contains all the elements they consider important.

3.2.3.3 Connection obscurity sub-characteristic

The connection obscurity sub-characteristic of modularity describes the degree to which an object oriented class or object module complies with the rule of explicit interfaces.



Connection Obscurity

The degree to which the connection from a module to an external element is unclear after examination of the module or the external element or both.

Connection obscurity is sub-characterised as variable connection, unstated relationship, distant connection and non-standard connection.

- Variable connection

  The nature of the connection from a module to external elements is not fixed.

- Unstated relationship

  The connection from a module to an external element is not explicitly stated within the module or the external element or both.

- Distant connection

  The connection is between a module and an external element with a distant, indirect relationship.

- Non-standard connection

  The connection from a module to an external element is unexpected and/or takes place via non-standard interfaces.

Non-standard connection is further sub-characterised as unexpected relationship and connection via non-standard interface.

- Unexpected relationship

  The relationship from a module to an external element is an unexpected one that users would not commonly look for and recognise.

- Connection via non-standard interface

  Connection between a module and an external element takes place outside the standard module interface.

Figure 3-10 Connection Obscurity sub-characterisation and conceptual definition

The explicit interface rule of modularity states that "Whenever two modules A and B communicate, this must be obvious from the text of A or B or both." (Meyer 1997, p.50). The term *explicit* implies that the connection between the modules must be clearly stated in the text of the modules. The situation where two modules share a global variable is a case of a connection or interface between modules that is not explicitly stated (Meyer 1997, p.50). Another example of a non-explicit interface is a child class accessing a distant ancestor through its immediate parents. In this case, the interface between the child and the distant ancestor is not explicitly stated in either the child or distant ancestor class modules. The sub-characteristic derived from the explicit interfaces rule is connection obscurity. Connection obscurity describes the degree to which the connections of a module are via obscure rather than explicit interfaces. From this view of the explicit interfaces rule, the following connection obscurity sub-characteristics are derived.

Variable connection describes the degree to which the connection between a module and external elements varies. The nature of the connection may not be resolved until the module is used within a software system, or until run-time, or it may vary during run-time. For example, in C++, a class can have an associated pointer to an object rather than an object instance declared as a member element. Depending on the software implementation, the number of objects that this pointer can refer to may vary during program execution from none to an unlimited number, thus obscuring the precise nature of the connection.

Unstated relationship refers to relationships of a module that are not explicitly defined. A connection between modules can be clearly discerned where a statement of the relationship between them is part of one module, or the other or both (Meyer 1997, p. 50). An example of a stated relationship is an object declared within a class. It is clear from the class declaration that it will be able to access the object's public interface elements. An example of an unstated relationship between modules is where two or more classes share access to a common global variable. Although the classes are connected via the global variable, the connection is not stated within either the class or global variable declarations.

Distant connection refers to non-immediate connections between modules. For example, if an object declared to be an immediate associated member of a class itself has a second associated object appearing in its interface, then the class is able to directly access the second object even though it does not have an immediate relationship with it. This represents a distant type of connection and contributes to connection obscurity.

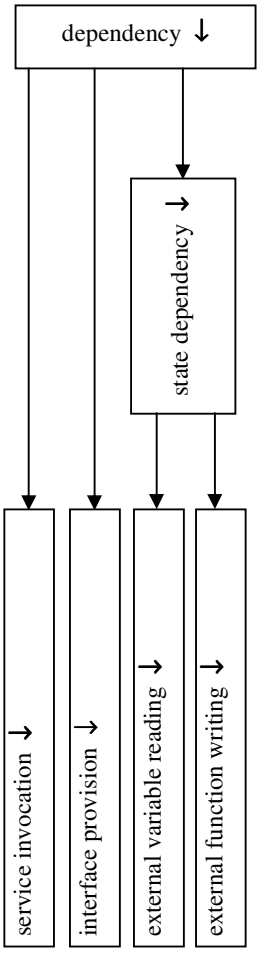
Non-standard connection described the degree to which the connection from a module to external elements is of an unexpected, uncommon or unusual type rather than of the standard, expected type. The non-standard connection sub-characteristic is further sub-characterised as unexpected relationship and connection via non-standard interface.

Unexpected relationship describes whether or not module relationships are of an unexpected non-standard nature. For example, an inheritance relationship between object oriented classes is a standard expected one however, the friend type relationship between classes of objects is not standard even though it is a feature of the C++ language.

Connection via non-standard interface is concerned with whether or not the connections of a module are via standard module interfaces. The module interfaces should be designed to protect their module and to provide a stable point of access to external elements. Connection via non-standard interface increases connection obscurity by creating a relationship between modules that is different to what would normally be expected. For example, in C++, a client object can normally only access the public and protected elements of a supplier object. If however the class from which the client object is instantiated is a friend to the supplier's class, the client object can access the normally inaccessible protected and private supplier elements.

3.2.3.4 Dependency sub-characteristic

The dependency sub-characteristic of modularity describes the degree to which an object oriented class or object module complies with the rule of small interfaces.



Dependency

The degree to which a module depends on external elements in order that it can perform its own functions correctly.

Dependency is sub-characterised as service invocation, interface provision and state dependency.

- Service invocation

  The dependence of a module on services provided by elements external to itself.

- Interface provision

  The dependency of a module on external modules to provide some or all of it's interface elements.

- State dependency

  The degree to which the correct preservation of a module's state is dependent on external elements.

State dependency is further sub-characterised as external variable reading and external function writing.

- External variable reading

  The degree to which the module's state is dependent on values contained in external variables.

- External function writing

  The degree to which the module's state is dependent on values written, by external functions, to attributes or external variables from which the module directly reads.

Figure 3-11 Dependency sub-characterisation and conceptual definition

The small interfaces rule of modularity states that "If two modules communicate, they should exchange as little information as possible." (Meyer 1997, p.48). This means that a module with high modularity operates as independently as possible of information coming from external elements. The small interfaces rule is described by the degree to which a module is dependent on information from external elements.

Service invocation describes the dependency of a module on services provided by elements external to the module. The module accesses these services by invoking class or object methods, or by invoking global functions. While a degree of service invocation is necessary to connect software modules into a functioning system, higher levels of modularity are achieved when module dependence on external service invocations is minimised.

Interface provision refers to a module's dependence on other modules to provide some or all of its interface elements. While such dependence may be necessary to promote other desirable software characteristics such as compatibility, it leads to reduced modularity in the dependent module because it requires the presence of another module or modules. An example of interface dependency is a child class with some inherited elements appearing directly in its public interface. The child class is dependent on the parent to provide interface elements to objects instantiated from the child.

State dependency describes the dependency of a module on external elements to ensure that the attributes and variables it reads from are maintained within a valid range. A module with high modularity reads values only from its own attributes and it is fully responsible for maintaining these values within a valid range. A module with reduced modularity directly reads values from attributes or global variables that are written to by other modules. It no longer has direct control over the validity of these values. The state dependency sub-characteristic is further sub-characterised as external variable reading and external function writing.

External variable reading describes the case of a module directly reading a value from an external variable. This could be a global variable or an attribute of another module. In this situation, the module is depending on the external attribute to be set at a value compatible with the modules operation.

External function writing describes the case of a function external to the module directly writing

a value to one of the module's internal attributes or writing a value to an external attribute or variable that the module directly reads from. In this situation, the module is dependent on the external function to ensure that one of the attributes it depends on is set to a valid value.

3.2.4 Validation of the sub-characterisation

The issue that validation of the sub-characterisation seeks to address is whether or not the selected sub-characteristics combine to provide an adequate description of modularity. Since the final modularity description obtained is based on this sub-characterisation, it is important to be satisfied that the sub-characterisation is adequate. At this stage of measure development, the validation of the sub-characterisation is by examination of the selected theory regarding the main characteristic of interest and the sub-characterisation and conceptual definitions arising from it. A subjective decision is then made regarding the extent to which the sub-characterisation identifies the aspects of the characteristic highlighted in the theory as having a significant effect on the levels of characteristic present in the software. In performing this assessment, both the sub-characteristic name and conceptual definition should be considered. Any ambiguities in the conceptual definitions should be corrected so that it is clear which particular aspects of the software are to be described by the developed measures.

3.3. Conclusion

Conceptual definition of the characteristic to be described by the measures is the first stage of the systematic measure development process. For a cluster type characteristic, the conceptual definition stage includes sub-characterisation of the main characteristic and the conceptual definition of each selected sub-characteristic. Prerequisite to sub-characterisation and conceptual definition is a theoretical understanding of the ways in which the characteristic to be described by the measures is manifest in the entity to be measured. The sub-characteristic and conceptual definition products of the conceptual definition stage are prerequisites to the entity modelling stage of descriptive measure development.

The aim of the systematic measure development process is to produce a set of measures that adequately describe the characteristic of interest. The conceptual definitions are an important stage of the measure development process because they communicate precisely the aspects of the software that these measures are intended to describe. The degree of detail with which a characteristic is conceptually defined will influence the degree of detail obtained from the description provided by the measures subsequently developed based on these conceptual

definitions. The conceptual definitions are also important to the eventual user of the measures because they convey the aspects of the software described by the set of measures, allowing the user to judge whether or not the measures are appropriate to their intended application. Should the user decide to proceed with measurement, the conceptual definitions also guide the analysis and interpretation of the measured data.

In this chapter, the process of software characteristic conceptual definition has been demonstrated for the characteristic of modularity of C++ class and object entities. Meyer's (1997) rules of object oriented software modularity provided the prerequisite theoretical basis of C++ class and object modularity conceptual definition. Modularity is a cluster characteristic (Gasking 1960; Ellis 1966) and so, the products of the demonstrated conceptual definition stage are a set of object oriented software modularity sub-characteristic with associated conceptual definitions. These are themselves prerequisite to the entity modelling stage of descriptive measure development presented in Chapter 4.

4. Entity Modelling

This chapter describes the entity modelling stage of software descriptive measure development. This stage corresponds to the shaded boxes in the Figure 4-1 diagrammatic representation of the measure development process. Section 4.1 of this chapter describes the natural language and mathematical entity modelling steps of entity modelling and section 4.2 demonstrates modelling of C++ class and object modularity, based on the sub-characterisation and conceptual definitions developed in section 3.2 of Chapter 3.

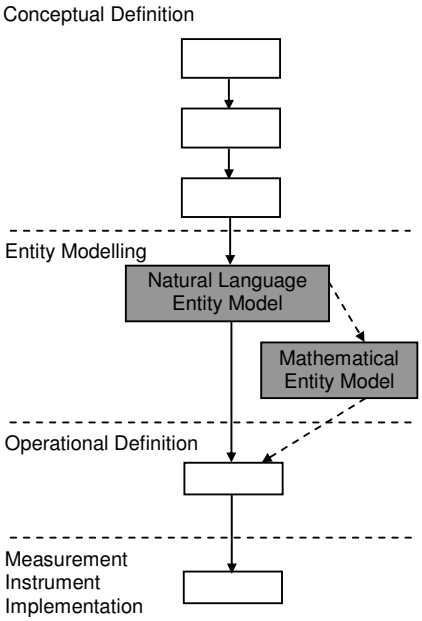


Figure 4-1 Entity modelling stage of the measure development process

4.1. Stage 2 of measure development process - entity modelling

The entity modelling stage of the systematic measure development process defines natural language and mathematical models of the software entity to be measured. Entity models together with conceptual definitions describe the theoretical basis from which descriptive measures are developed. Definition of the natural language entity model is an essential part of the measure development process while definition of an associated mathematical model is optional. Entity modelling aims to communicate an understanding of the characteristic to be described by the measures (Fenton 1994, p199). This understanding is intended to "reflect the specific viewpoint" (Fenton 1994, p199) from which the measures were developed. To effectively communicate an understanding, the natural language and mathematical entity models defined in this stage of the measure development process should contain sufficient detail to adequately describe the software while at the same time excluding unnecessary information.

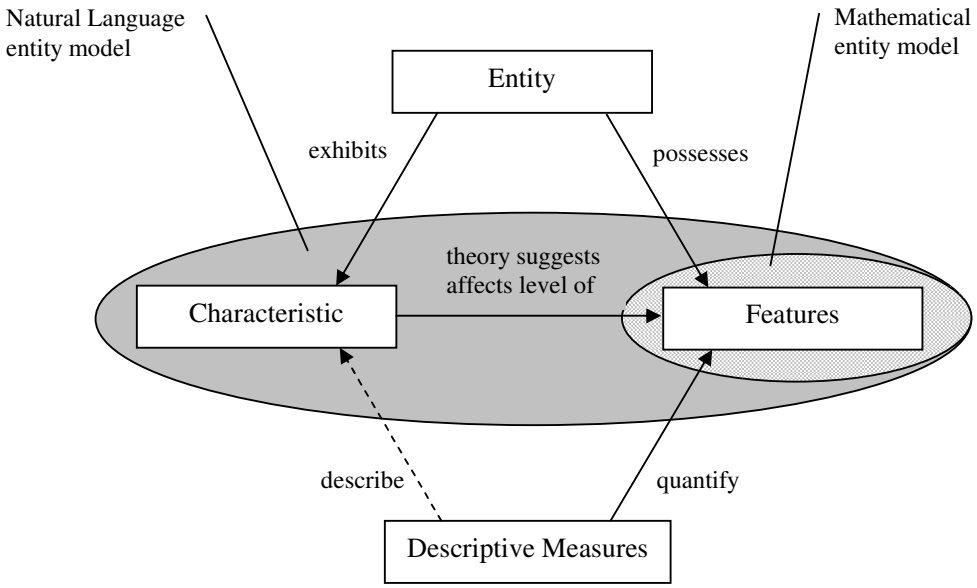


Figure 4-2 Natural language and mathematical entity models express the theoretical basis of descriptive measures

Figure 4-2 is an elaboration of the Figure 1-1 representation of the relationships between entities, characteristics, features and descriptive measures. Figure 4-2 shows that the natural language entity model expresses the theory that relates software features to software characteristics by identifying and describing the ways in which features of the software affect the level of characteristic present. When the characteristic of interest is a cluster (Gasking 1960; Ellis 1966, p34) that has been sub-characterised in the conceptual definition stage of measure development, the natural language model directly identifies software features affecting the levels of only the most refined sub-characteristics. The natural language entity model is important because it links the tangible features of the entity, quantified by the descriptive measures, to the entity characteristic of interest. This information facilitates the appropriate application of the measures as well as the subsequent analysis and interpretation of measured data. The identification of the features that affect the characteristic or sub-characteristics is guided by the selected theoretical basis of the measure development process. An inability to identify such features may indicate that a greater theoretical understanding of the characteristic or sub-characteristic is needed, in which case, further investigation may need to be undertaken to improve the theoretical basis of the measure development. Another possibility is that the characteristic or sub-characteristic is too broadly defined to allow the identification of the

particular features that affect it. In this case, the conceptual definition stage may need to be reiterated and the characteristic or sub-characteristic either more precisely defined or further sub-characterised and redefined.

The mathematical entity model is developed from the natural language entity model and supplements it by describing precisely the software features included in the natural language model. As Figure 4-2 shows, the mathematical model does not link these features to the characteristic and sub-characteristics of interest and so cannot be used as an entity model on its own. The advantage of having a mathematical model is that it provides an unambiguous description of the software and, if the type of model selected is suitable, the descriptive measures of the characteristic of interest can be similarly unambiguously defined in terms of the mathematical model. When selecting the type of mathematical model to use, it is important "to choose the model that most clearly emphasises the attribute(s) in question." (Fenton 1990, p178) Ideally, the mathematical model is able to describe all the features of the natural language model software description. If the mathematical model is unable to describe all the natural language model features, then the measures defined in terms of the mathematical model will also be unable to describe these features. In this situation it may be necessary to describe these omitted features with another type of mathematical model and define measures in terms of this new model. Alternatively, measures of the omitted features can be defined in natural language terms, taking care that their definition is unambiguous. A final alternative is to define no measures of the omitted features, recognising that this aspect of the natural language model will not be included in the software description obtained from the final set of measures.

Mathematical entity models have been used by several software measure development projects to provide a basis for measure definition. Briand, Daly and Wust (1999a) define a mathematical model to provide "a standardized terminology and formalism for expressing measures … which ensures that all measures using it are expressed in a fully consistent and operational manner." This model is then used to define measures of object oriented software coupling. (Briand, Daly & Wust 1999a, p 91). Bieman and Kang (1995) define a mathematical model of object oriented software and use it to define measures of cohesion. Abreu and Carapuca (1994) define a mathematical model of object oriented software with which to define measures of software quality. These mathematical models provide a formal expression of associated natural language models of the software and are intended to support the unambiguous definition of software measures.

The following sections 4.1.1, 4.1.2 and 4.1.3 describe the entity modelling stage of measure development in terms of its prerequisites, performance and products.

4.1.1 Prerequisites to the entity modelling stage

Prerequisites to the entity modelling stage are the conceptual definition of the characteristic to be described by the developed measures and an associated theoretical basis describing the features of the software that affect the level of characteristic present in the software. If the characteristic is a cluster (Gasking 1960; Ellis 1966, p34), then the conceptual definition stage will have produced a sub-characterisation and a set of conceptual definitions of these sub-characteristics. In this case, the most refined sub-characteristics with their associated conceptual definitions are prerequisite to the entity modelling stage. The entity modelling stage prerequisite theoretical basis must describe, for each of these sub-characteristics, features of the software that affect the level of sub-characteristic present.

4.1.2 Performance of the entity modelling stage

Once the prerequisite requirements have been met, the software entity models can be defined. Figure 4-3 describes the process of natural language entity modelling for a cluster (Ellis 1966, p34) type characteristic for which a set of sub-characteristics have been identified in the conceptual definition stage. The same process is applicable to natural language entity modelling of a non-cluster characteristic by considering the characteristic to be its own sub-characteristic. Each point of the natural language model should identify a specific feature of the software and describes how this feature affects the level of characteristic or sub-characteristic present. If the characteristic of interest is a cluster, then it may be practical to define separate natural language and mathematical models for each identified sub-characteristic.

A mathematical model can be defined based on the information contained within the natural language model. Firstly, the various features identified in the natural language model must be identified. These features may be components of the software or relationships between components. The mathematical model type should be selected on the basis that it is able to describe these features. More than one type of model may be needed to describe different software features, and some features may not be able to be described by any mathematical model.

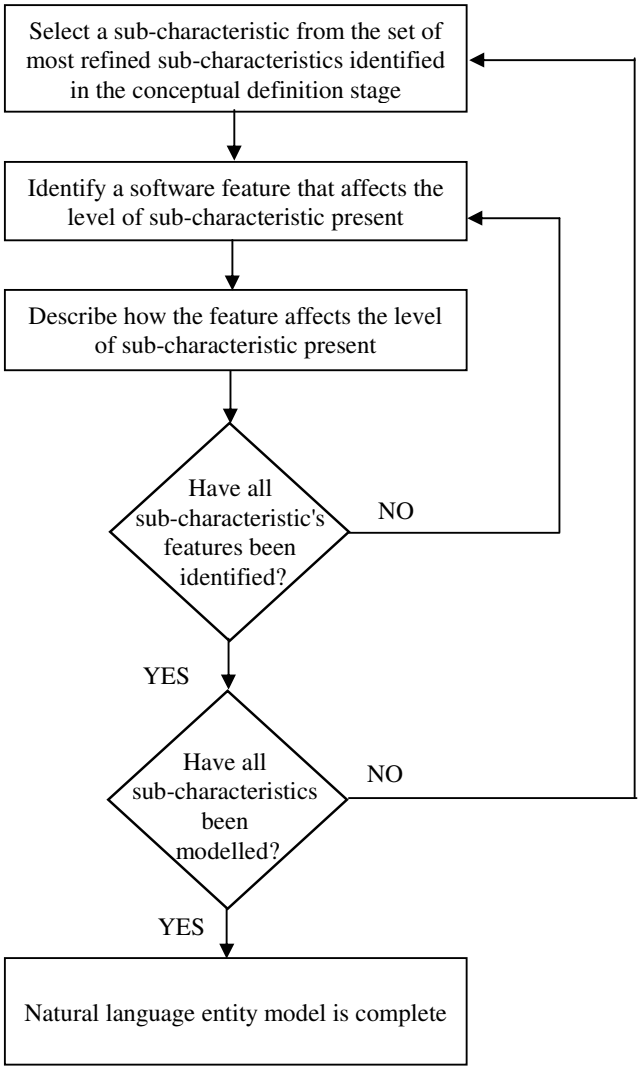


Figure 4-3 Process of natural language modelling of a software entity

4.1.3 Products of the entity modelling stage

The products of the entity modelling stage are a natural language model and an optional mathematical model of the software, describing the features of the software affecting the levels of characteristic present. If a mathematical model has been defined to describe the features identified in the natural language model then the validity of the description of the natural language model that it provides should be assessed.

4.1.4 Assessing mathematical model validity

As Figure 1-3 shows, where a mathematical model has been defined, the validity of the description it provides of the natural language model should be assessed. A fully valid mathematical model is able to completely describe all the software features identified in the natural language model. The validity of the mathematical model description of the natural language model is reduced by its inability to describe natural language model software features.

Determining the completeness of the mathematical model description of the natural language model involves examining the natural language model and comparing the software features identified in this model with the software features described by the mathematical model. Any omissions of the mathematical model should be corrected where possible. Where the mathematical model is unable to describe certain features of the software, then either

- the existing mathematical model can be discarded and a completely different type of mathematical model defined that is able describe all the important features of the software
- the existing mathematical model can be retained and supplemented by another type of model defined to describe the missing features or
- the shortcomings of the existing mathematical model can be explicitly identified, documented and accepted with the proviso that the measure application, analysis and interpretation phases of measurement may need to take into account the fact that some aspects of the characteristic of interest are not described by the measures developed based on this model. This option is most appropriate when the features of the software not described by the mathematical model occur infrequently in the software to be measured or when the user of the measures is not interested in obtaining a measured description of the software characteristic or sub-characteristic affected by these features.

4.1.5 Practical Considerations

The entity modelling stage of descriptive measure development aims to precisely identify the features of the software that the selected theoretical basis for measure development identifies as affecting the levels of characteristic of interest present in the software. When the theoretical basis from which the measures are being developed lacks sufficient detail, the measure developer may need to enhance the available theory in some way.

Abreu, Goulao and Esteves (1995) provide an example of extending a theoretical basis to accommodate more precise measure definition. The set of generic Metrics for Object Oriented Design (MOOD) measures of software quality developed by Abreu and Carapuca (1994) are tailored to the description of C++ quality by the definition of a set of "MOOD/C++ bindings" (Abreu, Goulao and Esteves 1995, pp. 47-51). These bindings extend the theoretical basis of the MOOD (Abreu and Carapuca 1994) measures by specifically identifying the features of C++ software that affect software quality. In a similar manner, an insufficiently detailed theoretical basis for measure development could be enhanced with reference to a new, more detailed software theory, with reference to an expert's understanding of the software, or with reference to the measure developers own understanding of the software. All these methods of theory enhancement are acceptable, as long as the natural language model of the software describes precisely the full theoretical basis used to develop the measures. Such a complete description allows a future user of the measures to examine the theoretical basis from which the measures were developed and determine whether or not this basis is appropriate for their own intended use.

The need to express, in the natural language software model, a detailed and precise understanding of the ways in which the characteristic of interest can lead to a large and complex entity model being developed. While the production of a large, detailed entity model may be a difficult and time consuming task requiring several iterations to achieve a satisfactory result, the quality of the description obtained from a set of measures is highly dependent on the quality of the entity model from which they were developed. A significant problem can arise when a natural language entity model cannot be fully described by a mathematical entity model. The advantage of the precise software description provided by a mathematical model may be offset by the disadvantage of the incomplete description obtained when the mathematical model cannot fully describe the natural language model. In a situation such as this, the decision to use a mathematical model must be based on the advantages of precision outweighing the disadvantages of missing information.

The following section 4.2 demonstrates the entity modelling stage of the development of descriptive measures of C++ class and object modularity.

4.2. Entity modelling of C++ class and object modularity

The entity modelling of C++ software is the second stage in the development of descriptive measures of class and object modularity. Entity model development is based on the modularity sub-characterisation and conceptual definitions developed in the previous stage, as well as on the selected measure development theoretical basis. The first step is to model the C++ class and C++ object entities. Once the C++ class and object entities have been modelled, the models describing class and object modularity sub-characteristics of interface dependence, external relationships, connection obscurity and dependency are defined.

The elements selected to be part of these initial base models influence the form of the subsequently defined modularity sub-characteristic models and hence, the form of the measure definitions. A study by Churcher and Shepperd (1995b, p. 71) provides an example of alternative entity models. Here, two different models of a class are described. One model includes inherited methods as part of the class model while the other considers them to be external to the class. The particular class model chosen affects the way that messages between class methods are assessed. The effect that the selected class model has on the final measure definitions can also be seen in the measures developed by Bieman and Kang (1995, p. 26). Here, the tight class cohesion (TCC) and loose class cohesion (LCC) measures have two different definitions depending on whether the abstracted class (AC) or local abstracted class (LAC) model is used as the foundation for the software model from which the measures are derived.

Section 4.2.3 describes the development of C++ class modularity entity models. Entity modelling of C++ class modularity is relatively straightforward as the entity-relationship type of mathematical models are able to fully describe their associated natural language models. This is not the case for the C++ object entity models described in section 4.2.4 because the entity-relationship mathematical models are unable to completely describe associated natural language models. The shortcomings of each object sub-characteristic mathematical model are discussed individually.

The entity models of class and object modularity include features directly related to classes and object-classes and features that are indirectly related to classes and object-classes through a direct relationship with class and object-class member methods and attributes. This is a

measure design decision made to allow greater flexibility in data analysis than would be provided by only identifying features at the direct class and object-class level. For example, in Section 4.2.3.2.1, feature 1.1.1 of the C++ class interface dependence natural language model is described with respect to individual class interface methods. To obtain a class level description, techniques such as calculating the mean, median or sum can be used. The individual method measures can also be analysed to produce a representation of a class similar to that of Figure 3-3. The measurement data collected at the method and attribute level can be analysed in different ways to provide varying descriptions of class or object-class level modularity.

4.2.1 Prerequisites

The prerequisites to the entity modelling stage of C++ class and object modularity measure development are a modularity sub-characterisation with associated conceptual definitions and a theoretical basis identifying the features of C++ software that affect the levels of modularity sub-characteristics present in the software. At the conceptual definition stage of measure development presented in Chapter 3, Meyer's (1997, pp. 46-53) rules of object oriented software modularity provided the theoretical basis of modularity sub-characterisation and conceptual definition. Meyer's (1997, pp. 46-53) rules of modularity will also provide part of the theoretical basis for the entity modelling stage. As discussed previously, the theoretical basis of entity modelling must identify features of the software that affect the levels of modularity present in the software. While Meyer's (1997, pp. 46-53) rules and their accompanying discussion indicate general software features that affect modularity, they do not identify particular C++ software features. To provide a sufficiently detailed theoretical basis for entity modelling, Meyer's (1997, pp. 46-53) rules must be interpreted with respect to C++ software features. This interpretation is documented within the natural language modularity entity models and is available for examination by a potential user of the measures. As mentioned in section 3.2.1 of Chapter 3, if the theoretical basis from which measures are developed is judged to be inadequate, a user is free to modify or discard it and similarly treat the measures developed from it.

4.2.2 Selection of the mathematical model

Entity-relation type mathematical models have been selected to describe the C++ software modularity features identified in the natural language models. The reasons for choosing this

type of model are that

- entity-relationship models are well understood by many people in the software development field and so provide a good means of communicating information.
- set based entity-relationship type models have been previously used to model object oriented software (Churcher and Shepperd 1995b) and to define object-oriented software measures (Briand, Daly and Wust 1997b; Briand, Daly and Wust 1999a).
- measures can be directly defined in terms of the sets of entity-relationship models and these definitions readily converted to relational database queries (Grassmann & Tremblay 1996, pp. 620-624).
- relational database packages are relatively inexpensive and readily available to run on a number of different platforms. For the prototype measurement instrument developed in this thesis, the Microsoft Access (Microsoft 1997) relational database application running on a PC is sufficient to implement and apply the measures.
- the Understand for C++ (Scientific Toolworks Inc. 2004) code parsing and data extraction application, identified as providing sufficient information to implement the majority of defined modularity measures, is based on a relational database model of object oriented software. Again, suitability, cost and availability, as well as compatibility with the selected database application influenced the choice of data collecting application and hence dictated the choice of mathematical model.

The style of entity-relationship diagram used in this thesis differs slightly from standard diagrams such as those described by Hawryszkiewycz (1990), in that relationships are shown as rectangles rather than diamonds. To distinguish between entities and relationships, entities are shown as large rectangles with the entity name written in uppercase and relationships are shown in smaller rectangles with the relationship name written in mainly lowercase. Entity-relationship models are not the only types of model that could be used. In particular, the unified modelling language (UML) defines several different types of object oriented software models (Eriksson and Penker 1998) which may be appropriate to software measure development. One direction for possible further research arising from this thesis is the investigation of different types of mathematical entity models and their suitability as a basis for measure development and collection. For instance, Tang and Chen (2002) describe the collection of Chidamber and Kemerer's (1994) measures from UML class, collaboration and activity diagrams and Arisholm, Briand and Foyen (2004) use UML class and sequence diagram to model object oriented software dynamic coupling.

In this thesis, the mathematical entity-relationship model is defined as an implementation rather than as a conceptual model. This is a result of the research method used to formulate the systematic process of measure development which was to develop measures in a practical way and from this experience define the process. One effect of this is that mathematical model design and implementation decisions have been made that in some cases mean that some of the measures defined in natural language terms cannot also be defined in mathematical terms due to limitations of the mathematical model implementation. In the tables defining the modularity measures, comments are included to indicate where this situation arises. The limitations of the mathematical models flow on to the implementation of the measurement instrument, restricting the measures that are included in the measurement instrument. Were a different mathematical model defined, then it is possible that more natural language measures could also be defined in mathematical terms and implemented in a measurement instrument. A possible approach would be to define the mathematical model at the entity modelling stage in conceptual terms, define the measures in these terms, and then redefine the mathematical models and measure definitions in implementation specific terms in the measurement instrument stage where they could be tailored to the particular strengths and limitations of the selected measurement instrument. An area for possible future research would be to take the example of the implementation specific mathematical model defined in this thesis and from it develop a more general, conceptual model. This conceptual model could then form the basis of revised mathematical measure definitions.

4.2.3 C++ class modularity

With the necessary prerequisites identified and mathematical model selected, entity modelling of C++ class and object modularity can proceed. In Chapter 3, the conceptual definition of C++ class modularity identified interface dependence, external relationships, connection obscurity and dependency as important modularity sub-characteristics. In this entity modelling chapter, section 4.2.3.1 defines preliminary natural language and mathematical models of the C++ class module. Based on this class model, sections 4.2.3.2, 4.2.3.3, 4.2.3.4 and 4.2.3.5 define entity models for each of the identified modularity sub-characteristics.

4.2.3.1 C++ class module model

The C++ class interface dependence, external relationships, connection obscurity and dependency entity models describe the modularity of a C++ class module. As discussed in

section 4.2, before defining these models it is first necessary to model a C++ class entity.

4.2.3.1.1 Natural language model of C++ class module

The following points define a natural language model of a C++ class describing the elements that constitute a C++ class module. The C++ class mathematical model defined in section 4.2.3.1.2 describes the features identified in this natural language model. Included with each natural language model point are the names of the associated mathematical model sets describing each feature. A one to one correspondence between natural language feature and mathematical model set indicates that the mathematical model is able to describe the associated feature.

- A class (C) has member method (MM, M) and member attribute (MA, A) elements. Methods and attributes that a class can directly access through inheritance are not considered to be members of the class. Object instances of other classes are not members of a class. Others to exclude object instances from the class model include Abreu and Melo (1996, p.92) and Bieman and Kang (1995, p.206).
- Member attributes (MA, A) are instances of, or pointers to, a C++ primitive data types, declared within the class (C) definition. A primitive data type is one of char, double, float, int, long, short, signed or unsigned.
- Member methods (MM, M) are the functions and procedures declared within the class (C) definition.
- A class (C) is divided into interface and hidden regions. Interface elements have a public or protected level of protection (MM, M, MA, A) and hidden elements have a private level of protection (MM, M, MA, A).

Figure 4-4 illustrates a C++ class module.

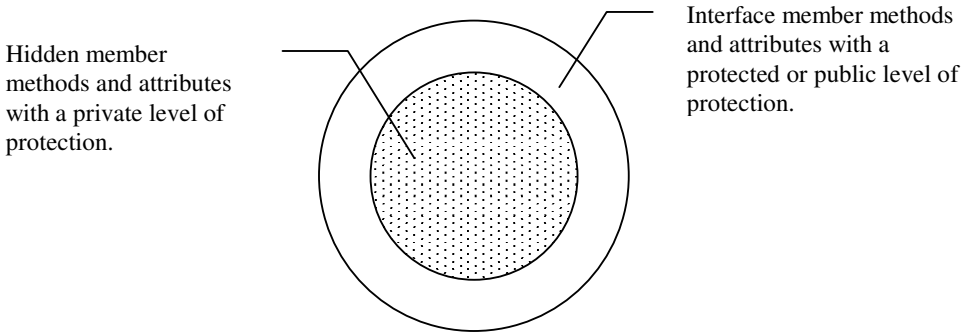


Figure 4-4 C++ class interface and hidden elements

4.2.3.1.2 Mathematical model of C++ class module

The entity-relationship model describing the member elements of a C++ class is illustrated in Figure 4-5. The set definitions of the entities and relationships of this class model are detailed in Appendix 1. As indicated in the class natural language model definition, the C++ class mathematical model is able to describe all the features of the C++ class natural language model.

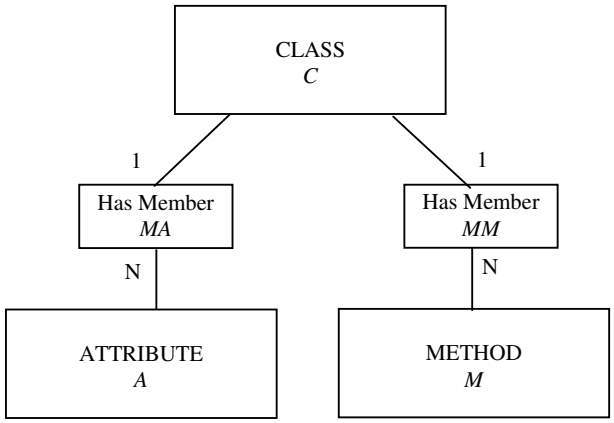


Figure 4-5 C++ class mathematical model

This model of a C++ class module forms the core of the following models describing class interface dependence, external relationships, connection obscurity and dependency.

4.2.3.2 Interface dependence sub-characteristic of C++ class modularity

In section 3.2.3.1, interface dependence is conceptually defined as the degree to which the module interface elements are dependent on each other and on the implementation specific details of the module. Interface element interdependence, interface size and data exposure are identified as the most refined interface dependence sub-characteristics. The following natural language and mathematical models describe features of C++ software that increase the levels of interface element interdependence, interface size and data exposure present in a class module, thereby reducing its modularity. Natural language model features are followed by their associated mathematical model set name, shown in brackets. A one to one correspondence between natural language feature and mathematical model set indicates that the mathematical model is able to describe the associated feature.

4.2.3.2.1 Natural language model of C++ class interface dependence

1. Interface Dependence : C++ class module
   - 1.1 Interface Element Interdependence - The dependency producing connections between interface elements within the same module.
     - 1.1.1 Class (C) interface methods (MM, M) directly read (MCReadA) and/or write (MCWriteA) same class (C) interface attributes (MA, A).
       Interface methods are directly dependent on interface attributes. Should an interface attribute be modified, there is a chance that interface methods dependent on it will need to be modified too.
     - 1.1.2 Class (C) interface methods (MM, M) indirectly read (MICReadA) and/or write (MICWriteA) same class (C) interface attributes (MA, A).
       Interface methods are indirectly dependent on interface attributes. Should an interface attribute be modified, there is a chance that interface methods dependent on it will need to be modified too.
     - 1.1.3 Same class (C) interface methods (MM, M) directly invoke (MCInvM) each other.
       Interface methods are directly dependent on other interface methods. Should a method be modified, there is a chance that other interface methods dependent on it will need to be modified too.
     - 1.1.4 Same class (C) interface methods (MM, M) indirectly invoke (MICInvM) each other.
       Interface methods are indirectly dependent on other interface methods. Should a method be modified, there is a chance that other interface methods dependent on it will need to be modified too.

1.2 Interface Size - The size of a module's interface.

1.2.1 Class (C) has a relatively high proportion of total member attributes (MA, A) in the interface.

Implementation specific details of the class are included in the interface in the form of class attributes.

1.2.2 Class (C) has a relatively high proportion of total member methods (MM, M) in the interface.

Methods that are likely to contain implementation specific details of the class are included in the interface.

1.2.3 Class (C) interface methods (MM, M) are relatively large having:

1.2.3.1 many lines of code (M)

Code that implements the class services is likely to be contained within the interface methods.

1.2.3.2 many same class (C) method (MM, M) invocations (MCInvM).

Implementation specific multiple method invocations are contained within the interface methods.

1.2.3.3 many same class (C) direct attribute (MA, A) accesses (MCReadA, MCWriteA).

Implementation specific direct attribute accesses are contained within the interface methods.

1.3 Data Exposure - The degree to which the module data is revealed to the module interface.

1.3.1 Class (C) has attributes (MA, A) in the interface.

Class attributes, highly implementation specific elements of the class, occur in the class interface.

1.3.2 Class (C) individual interface methods (MM, M) directly read (MCReadA) and/or write (MCWriteA) same class (C) attributes (MA, A).

The more attributes an interface methods accesses, the greater the chance that it will be affected by an attribute change.

1.3.3 Class (C) individual member attributes (MA, A) directly read (MCReadA) and/or written (MCWriteA) by same class (C) interface methods (MM, M).

The more interface methods directly access a class attribute, the greater that chance that a change to that attribute will affected interface methods operation.

4.2.3.2.2 Mathematical model of C++ class interface dependence

The entity-relationship model of C++ class interface dependence is illustrated in Figure 4-6. Methods within a class module are able to directly invoke other same class methods and directly read and write same class attributes. Class-read, class-write and class-invoke describe these relationships in the mathematical model. The set definitions of the entities and relationships of this interface dependence model are detailed in Appendix 1. As indicated in the interface dependence natural language model definition, this mathematical model is able to describe all the features of the natural language model.

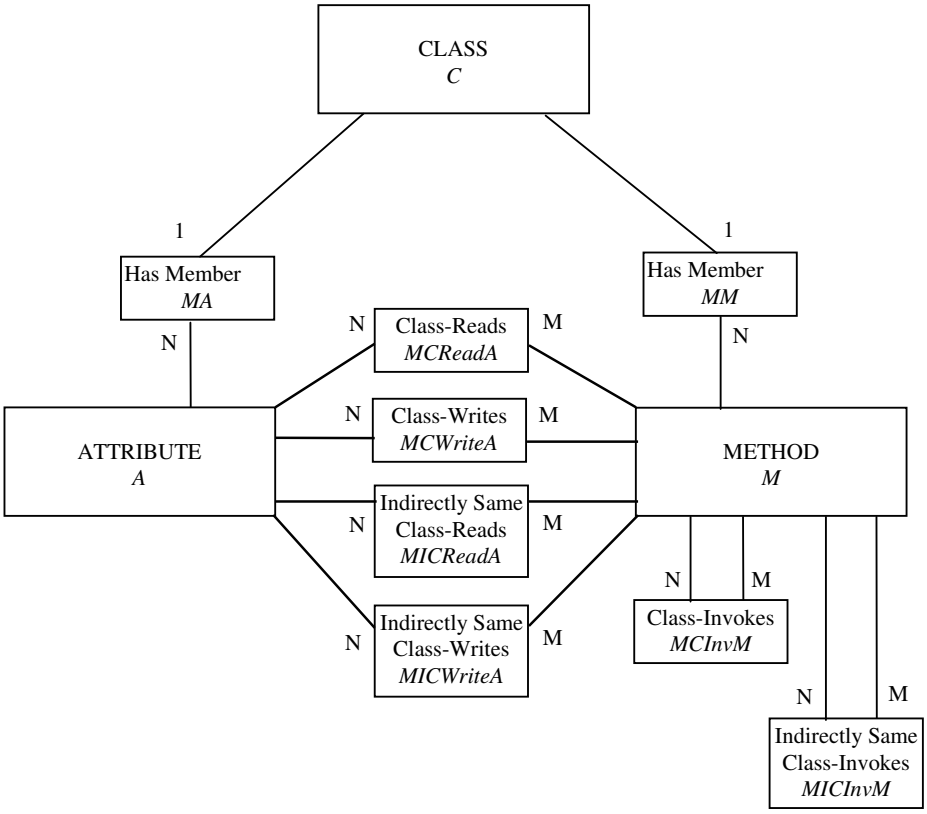


Figure 4-6 C++ class interface dependence mathematical model

4.2.3.3 External relationships sub-characteristic of C++ class modularity

In section 3.2.3.2, external relationships is conceptually defined as the degree to which a module has relationships with elements external to itself. External relationships within the system is identified as the most refined external relationships sub-characteristic. The following natural language and mathematical models describe features of C++ software that increase the levels of external relationships of a class module, thereby reducing its modularity. Natural

language model features are followed by their associated mathematical model set name, shown in brackets. A one to one correspondence between natural language feature and mathematical model set indicates that the mathematical model is able to describe the associated feature.

4.2.3.3.1 Natural language model of C++ class external relationships

2. External Relationships : C++ class module
   2.1 Within the System - The relationships between a module and elements external to the module but still within the given software system.
      2.1.1 Class (C) has one or more immediate parent (IP) classes (C).
      The class has an inheritance relationship with one or more immediate parent classes.
      2.1.2 Class (C) has one or more distant ancestor (IDA) classes (C).
      The class has an inheritance relationship with one or more distant ancestor classes.
      2.1.3 Class (C) has one or more immediate child (IP) classes (C).
      The class has an inheritance relationship with one or more immediate child classes.
      2.1.4 Class (C) has one or more distant descendent (IDA) classes (C).
      The class has an inheritance relationship with one or more distant descendent classes.
      2.1.5 Class (C) has one or more immediate friend (CEF) classes (C).
      One or more other classes have friend privileges of access to the class.
      2.1.6 Class (C) has one or more immediate friend (FF) global functions (F).
      One or more global functions have friend privileges of access to the class.
      2.1.7 Class (C) is friend (CEF) to one or more other classes (C).
      The class has friend privileges of access to one or more other classes.
      2.1.8 Class (C) has one or more global functions (F) within its scope (SF)
      Within the measured software system there are global functions that the class can directly access.
      2.1.9 Class (C) has one or more global variables (V) within its scope (SV)
      Within the measured software system there are global variables that the class can directly access.

4.2.3.3.2 Mathematical model of C++ class external relationships

The entity-relationship model of C++ class external relationships is illustrated in Figure 4-7. The set definitions of the entities and relationships of this external relationship model are detailed in Appendix 1. This mathematical model is able to describe all the features of the class module natural language model.

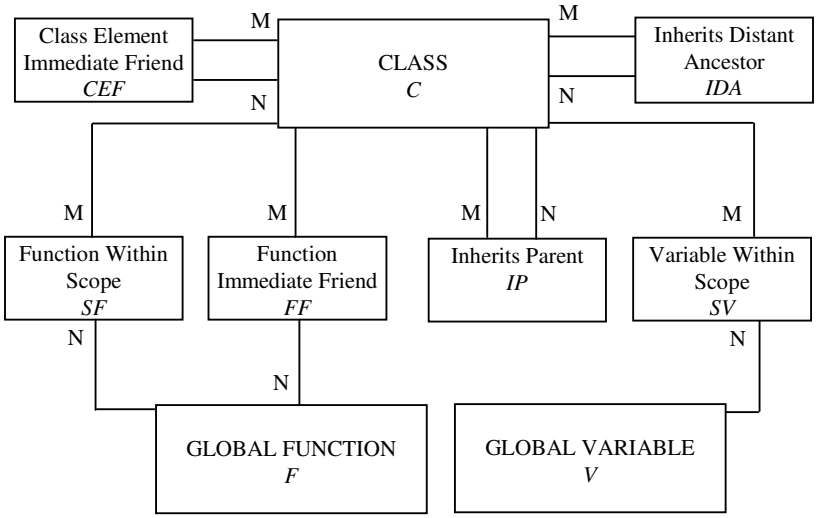


Figure 4-7 C++ class external relationships mathematical model

4.2.3.4 Connection obscurity sub-characteristic of C++ class modularity

In section 3.2.3.3, connection obscurity is conceptually defined as the degree to which the connection from a module to an external element is unclear after examination of the module or the external element or both. Variable connection, unstated relationship, distant connection, unexpected relationship and connection via non-standard interface are identified as the most refined connection obscurity sub-characteristics. The following natural language and mathematical models describe features of C++ software that increase the levels of variable connection, unstated relationship, distant connection, unexpected relationship and connection via non-standard interface of a class module, thereby reducing its modularity. Natural language model features are followed by their associated mathematical model set name, shown in brackets. A one to one correspondence between natural language feature and mathematical model set indicates that the mathematical model is able to describe the associated feature.

4.2.3.4.1 Natural language model of C++ class connection obscurity

3. Connection Obscurity : C++ class module

3.1 Variable Connection - The nature of the connection from a module to external elements is not fixed.

This sub-characteristic is not applicable to C++ classes.

3.2 Unstated Relationship - The connection from a module to an external element is not explicitly stated within the module or the external element or both.

3.2.1 Class (C) member methods (MM, M) directly accesses (MGReadV, MGWriteV) one or more global variables (V).

The relationship between a class and a global variable can only be discerned from examination of the class member methods that directly read or write the global variable. This relationship cannot be discerned from examination of the class declaration or the global variable declaration.

3.2.2 Class (C) member methods (MM, M) directly invoke (MGInvF) one or more global functions (F).

The relationship between a class and a global function can only be discerned from examination of the class member methods that directly invoke the global function. This relationship cannot be discerned from examination of the class declaration or the global function declaration.

3.3 Distant Connection - The connection is between a module and an external element with a distant, indirect relationship.

3.3.1 Class (C) member methods (MM, M) directly access (MGReadV, MGWriteV) a global variable (V) that is also directly accessed (MGReadV, MGWriteV) by member methods (MM, M) of another class (C).

A class is indirectly connected to another class if they share access to a common global variable by reading from it or writing to it or both.

3.3.2 Class (C) member methods (MM, M) directly access (MCReadA, MCWriteA, MCInvM) the member methods and/or attributes (MM, M, MA, A) of a distant ancestor (IDA) class (C).

A class can access the visible elements of its own distant ancestors. The origin and nature of distant ancestor member elements is more difficult to determine than that of immediate parent elements and hence the nature of the connection is obscured.

3.4 Unexpected Relationship - The relationship from a module to an external element is an unexpected one that users would not commonly look for and recognise.

3.4.1 Class (C) member methods (MM, M) directly access (MGReadV, MGWriteV) a global variable (V) that is also directly accessed (MGReadV, MGWriteV) by member methods (MM, M) of another class (C).

A connection between classes due to a common global variable is an unexpected relationship that is not specified in the class declaration.

3.4.2 Class (C) has one or more immediate friend (CEF) classes (C).

Friendship is not the standard relationship between classes.

3.4.3 Class (C) is friend (CEF) to other classes (C).

Friendship is not the standard relationship between classes.

3.4.4 Class (C) has one or more friend (FF) global functions (F).

Friendship is not the standard relationship between classes and global functions.

3.5 Connection via Non-standard Interface - Connection between a module and an external element takes place outside the standard module interface.

3.5.1 Class (C) member methods (MM, M) directly access (MGReadV, MGWriteV) a global variable (V) that is also directly accessed (MGReadV, MGWriteV) by member methods (MM, M) of another class (C).

A global variable is not part of the standard interface of a class and yet it forms a connection between the classes that access it.

3.5.2 Class (C) member methods (MM, M) directly access (MCReadA, MCWriteA) the member attributes (MA, A) of another class (C).

A standard class interface does not contain attributes.

3.5.3 Class (C) member attributes (MA, A) directly accessed (MCReadA, MCWriteA) by member methods (MM, M) of other classes (C).

A standard class interface does not contain attributes.

4.2.3.4.2 Mathematical model of C++ class connection obscurity

The entity-relationship model of C++ class connection obscurity is illustrated in Figure 4-8. The set definitions of the entities and relationships of this connection obscurity model are detailed in Appendix 1. This mathematical model is able to describe all the features of the class connection obscurity natural language model.

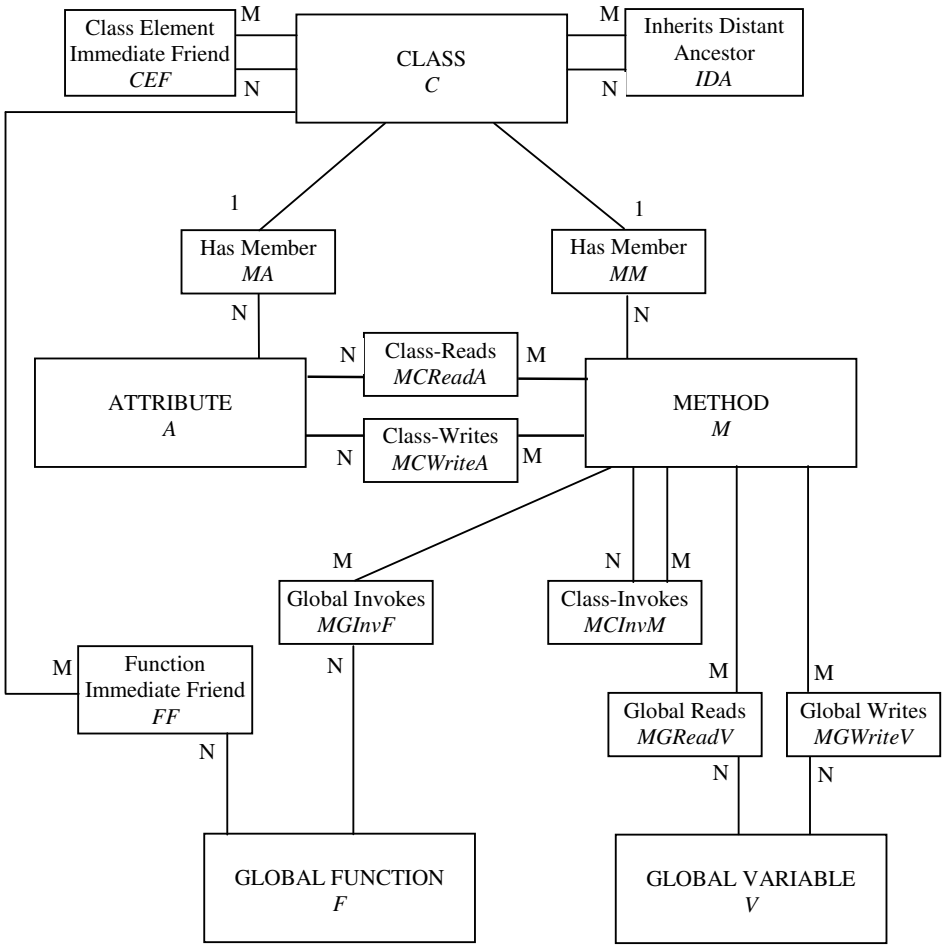


Figure 4-8 C++ class connection obscurity mathematical model

4.2.3.5 Dependency sub-characteristic of C++ class modularity

In section 3.2.3.4, dependency is conceptually defined as the degree to which a module depends on external elements in order that it can perform its own functions correctly. Service invocation, interface provision, external variable reading and external function writing are identified as the most refined dependency sub-characteristics. The following natural language and mathematical models describe features of C++ software that increase the levels of service invocation, interface provision, external variable reading and external function writing present in a class module, thereby reducing its modularity. Natural language model features are followed by their associated mathematical model set name, shown in brackets. A one to one correspondence between natural language feature and mathematical model set indicates that the mathematical model is able to describe the associated feature.

4.2.3.5.1 Natural language model of C++ class dependency

4. Dependency : C++ class module
   - 4.1 Service Invocation - The dependence of a module on services provided by elements external to itself.
     - 4.1.1 Class (C) member methods (MM, M) directly invoke (MGInvF) one or more global functions (F).
       To perform its own tasks, a class is dependent on the services provided by a global function.
     - 4.1.2 Class (C) member methods (MM, M) directly invoke (MCInvM) member methods (MM, M) of other classes (C).
       To perform its own tasks, a class is dependent on the services provided by another class.
   - 4.2 Interface Provision - The dependency of a module on external modules to provide some or all of its interface elements.
     This sub-characteristic is not applicable to C++ classes.
   - 4.3 External Variable Reading - The degree to which the module's state is dependent on values contained in external variables.
     - 4.3.1 Class (C) member methods (MM, M) directly read (MGReadV) one or more global variables (V).
       To perform its own tasks, a class is dependent on state information provided by a global variable.

4.3.2 Class (C) member methods (MM, M) directly read (MCReadA) from one or more member attributes (MA, A) of another class (C).

To perform its own tasks, a class is dependent on state information provided by an attribute of another class.

4.4 External Function Writing - The degree to which the module's state is dependent on values written, by external functions, to attributes or external variables from which the module directly reads.

4.4.1 Class (C) member methods (MM, M) directly read (MGReadV) from a global variable (V) directly written to (MGWriteV) by the member methods (MM, M) of another class (C).

The class relies on other classes to maintain a global variable in a state compatible with its own correct operation.

4.4.2 Class (C) member methods (MM, M) directly read (MGReadV) from a global variable (V) directly written to (FWriteV) by a global function (F).

The class relies on a global function to maintain a global variable in a state compatible with the class's own correct operation

4.4.3 Class (C) member methods (MM, M) directly read (MCReadA) from a static attribute (A) that is directly written to (MCWriteA) by the member methods (MM, M) of another class (C).

The class relies on other classes to maintain a static attribute with a value compatible with its own correct operation.

4.4.4 Class (C) member attribute (MA, A) is directly written to (MCWriteA) by a member method (MM, M) of another class (C).

The class relies on another class to maintain one of its own attributes with a value compatible with its own correct operation.

4.2.3.5.2 Mathematical model of C++ class dependency

The entity-relationship model of C++ class dependency is illustrated in Figure 4-9. The set definitions of the entities and relationships of this dependency model are detailed in Appendix 1. This mathematical model is able to describe all the features of the class dependency natural language model.

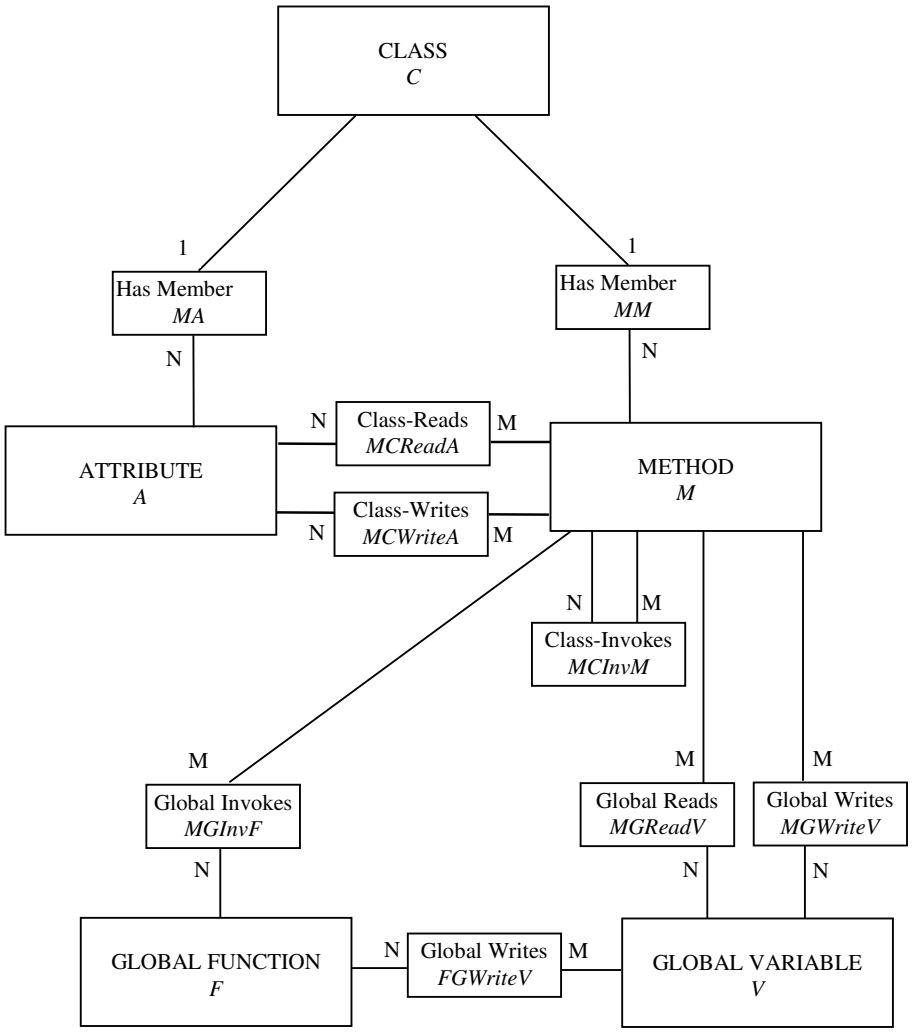


Figure 4-9 C++ class dependency mathematical model

4.2.3.6 Class modularity sub-characteristics sharing identical feature descriptions

Several modularity sub-characteristics are listed in the natural language model of C++ class modularity with the same features affecting their presence in the software. Table 4-1 lists these sub-characteristics with their associated common features.

Class Modularity Sub-characteristics	Feature
Interface Dependence : Interface Implementation Dependence: Interface Size Interface Dependence : Interface Implementation Dependence: Data Exposure	1.2.1 1.3.1
Interface Dependence : Interface Implementation Dependence: Interface Size Interface Dependence : Interface Implementation Dependence: Data Exposure	1.2.3.3 1.3.2
External Relationships Connection Obscurity : Non-standard Connection : Unexpected Relationship	2.1.5 3.4.1
External Relationships Connection Obscurity : Non-standard Connection : Unexpected Relationship	2.1.6 3.4.4
External Relationships Connection Obscurity : Non-standard Connection : Unexpected Relationship	2.1.7 3.4.3
Connection Obscurity : Unstated Relationship Dependency : Service Invocation	3.2.2 4.1.1
Connection Obscurity : Distant Connection Connection Obscurity : Non-standard Connection : Unexpected relationship Connection Obscurity : Non-standard Connection : Connection via Non-standard Interface	3.3.1 3.4.1 3.5.1

Table 4-1 Class modularity sub-characteristics sharing common natural language model features

The fact that several modularity sub-characteristics share common features implies that these features influence different aspects of modularity and have a stronger overall impact on general class modularity than features that only affect a single characteristic of modularity. Having some features occur more than once in the modularity sub-characterisation provides a natural weighting of these features, meaning that they have a greater influence over the final measured level of modularity determined for each class. On the other hand, it could be argued that some characteristics sharing several features in common should be combined into a single characteristic with only a single occurrence of the common features. For example, in Table 4-1, the interface size and data exposure sub-characteristics of interface dependence share several features in common and could be combined into a single interface implementation dependence

sub-characteristic. In this thesis, it has been decided to keep these sub-characteristics separate as they still have several exclusive features however future work may indicate that they are best combined.

Where high level sub-characteristics have features in common, dependencies are introduced that may affect the types of data analysis that can be performed on the measurement data. For example, Table 4-1 shows that the external relationships and connection obscurity sub-characteristics of class modularity have three different features in common. Were a factor analysis performed on the measurement data of the four high level modularity sub-characteristics, these features in common would reduce the discriminating power of the external relationships and connection obscurity sub-characteristics. One strategy to overcome this problem would be to remove all measures describing software features in common between modularity sub-characteristics. In this way, the remaining measures would describe the different sub-characteristics in dissimilar terms by quantifying dissimilar features. To identify measures describing common features so that they can be removed from the measured description for some analysis procedures, Table 5-14 lists the measures quantifying common software features of C++ class modularity.

4.2.4 C++ object modularity

As for class modularity, the Chapter 3 conceptual definition of C++ object modularity identified interface dependence, external relationships, connection obscurity and dependency as important modularity sub-characteristics. In this entity modelling chapter, section 4.2.4.1 defines preliminary natural language and mathematical models of the C++ object module. Based on this object model, sections 4.2.4.2, 4.2.4.3, 4.2.4.4 and 4.2.4.5 then define object entity models for each object modularity sub-characteristic. The differences between objects and classes, as they are modelled in this thesis, are due to the different elements from which they are considered to be composed and the different type of relationships in which they participate.

4.2.4.1 C++ object module model

The C++ object interface dependence, external relationships, connection obscurity and dependency entity models describe the modularity of a C++ object module. As discussed in section 4.2, before defining these models it is first necessary to model a C++ object module

entity. The object model differs from the class model in that an object's elements are comprised of the member methods and attributes of the class from which it is instantiated, plus all the member methods and attributes of all the ancestors to this class. Also, when compared to a class module, different levels of protection identify object hidden and interface regions. The following natural language and mathematical models describe the structure of an object module. Each object instantiated from a single class has the same module structure and so, rather than describe the modularity of each object in a software system, the object type modularity of each class is described. The term object-class is used to refer to the object type modularity associated with a class.

4.2.4.1.1 Natural language model of C++ object module

The following points define a natural language model of a C++ object describing the elements that constitute a C++ object module. The C++ object mathematical model defined in section 4.2.4.1.2 describes the features identified in this natural language model. Natural language model features are followed by their associated mathematical model set name, shown in brackets. A one to one correspondence between natural language feature and mathematical model set indicates that the mathematical model is able to describe the associated feature.

- An object-class (C) has member methods and attributes (MM, M, MA, A), accessible methods and attributes (AM, M, AM, A) and inaccessible methods and attributes (IAM, M, IAA, A). The member methods and attributes of the object-class (C) are the member methods and attributes of the same class (C). The accessible methods and attributes (AM, M, AA, A) are the member methods and attributes (MM, M, MA, A) of ancestor classes (IP, IDA) to the class (C) that object-class (C) can directly access. The inaccessible methods and attributes (IAM, M, IAA, A) are the member methods and attributes (MM, M, MA, A) of ancestor classes (IP, IDA) to the class (C) that object-class (C) cannot directly access. In a similar manner, the unified modelling language (UML) description of the class generalisation-specialisation relationship recognises that some object elements are inherited but are inaccessible to the object (Eriksson and Penker 1998, p. 94).

- Object-class (C) member attributes (MA, A) are instances of, or pointers to, a C++ primitive data types, declared within the class (C) definition where class (C) is the same as object-class (C).

- Object-class (C) member methods (MM, M) are the functions and procedures declared within the class (C) definition where class (C) is the same as object-class (C).
- Object-class (C) accessible and inaccessible attributes (AA, IAA, A) and accessible and inaccessible methods (AM, IAM, M) are member elements of immediate parent (IP) and distant ancestor (IDA) classes (C) to the object-class (C).
- An object-class (C) is divided into interface and hidden regions. Interface elements have a public level of protection (MM, AM, IAM, M, MA, AA, IAA, A) and hidden elements have a private, protected or inaccessible level of protection (MM, AM, IAM, M, MA, AA, IAA, A).
- The levels of protection assigned to the elements of a C++ object-class are determined according to the following rules.
  - Object-class elements that are member elements of the same class as the object-class have the same level of protection within the object-class as they had within the class.
  - The levels of protection of inherited elements within an object-class are determined by the levels of protection of the elements within the immediate parent class and the level of protection assigned to the inheritance. Table 4-2 summarises the rules of level of protection assignment within the C++ object-class model.

Immediate parent class inheritance level of protection	Element level of protection within immediate parent class	Resultant level of protection within object-class
public	public	public
	protected	protected
	private	inaccessible
	inaccessible	inaccessible
protected	public	protected
	protected	protected
	private	inaccessible
	inaccessible	inaccessible
private	public	private
	protected	private
	private	inaccessible
	inaccessible	inaccessible

Table 4-2 Rules for the assignment of C++ object-class element levels of protection

Figure 4-10 illustrates a C++ object-class module.

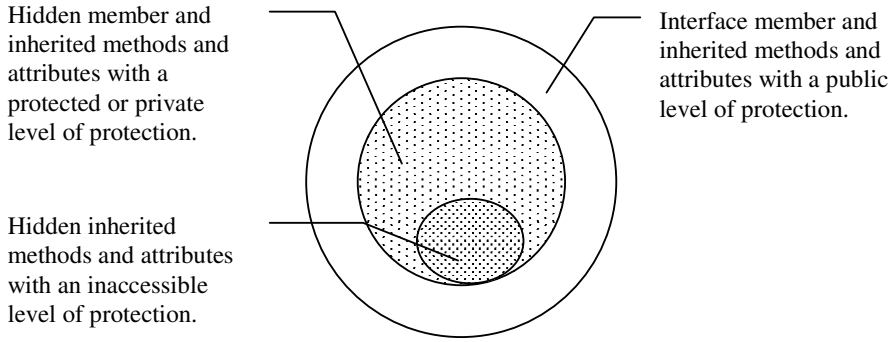


Figure 4-10 C++ object-class interface and hidden elements

4.2.4.1.2 Mathematical model of C++ object module

The entity-relationship model describing the member elements of a C++ object is illustrated in Figure 4-11. The set definitions of the entities and relationships of this object model are detailed in Appendix 1. As discussed in section 4.2.4.1.3, situations exist in which this mathematical model is unable to fully describe all the features of the object module natural language model.

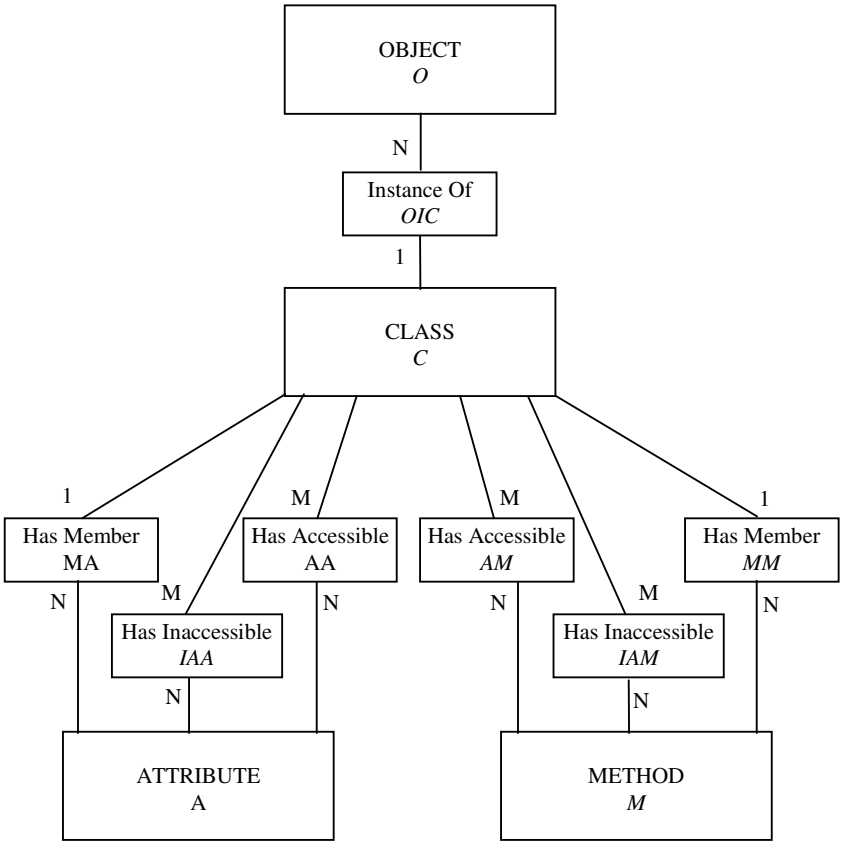


Figure 4-11 C++ object-class mathematical model

4.2.4.1.3 C++ object module mathematical model completeness

The entity-relationship model of a C++ object-class represents a simplification of the real-world object-class described in the natural language model. The AM, IAM, AA and IAA relations of the Figure 4-11 entity-relationship model of an object-class are unable to describe all the object-class's methods and attributes when the object-class inherits more than once from the same ancestor class. An ancestor class that is inherited more than once contributes multiple occurrences of its methods and attributes to the resulting object-class however, the AM, IAM, AA and IAA relations are only able to record a single occurrence of these inherited attributes and methods.

Multiple inheritance of a class is not a common occurrence in object oriented software so for the majority of object-classes, the entity-relationship model of Figure 4.11 provides a

description of all inherited attributes and methods and therefore provides a complete description of the natural language object model. In the situation where the mathematical model is unable to completely describe a C++ object-class due to its inheritance of duplicate elements from an ancestor class, only a single inheritance of the elements will be described by the mathematical model. Measures of affected object-classes using the AM, IAM, AA and IAA sets as part of their definition will have reduced validity. The full implication of this for the measured description of modularity of an affected object-class is discussed in detail in section 5.2.4. Figure 4-12 illustrates the problems associated with multiple inheritance of a class and the modelling compromise adopted.

Problem : Class A inherits methods and attributes from two versions of Class E

Solution : Record only the methods and attributes inherited from a single version of Class E.

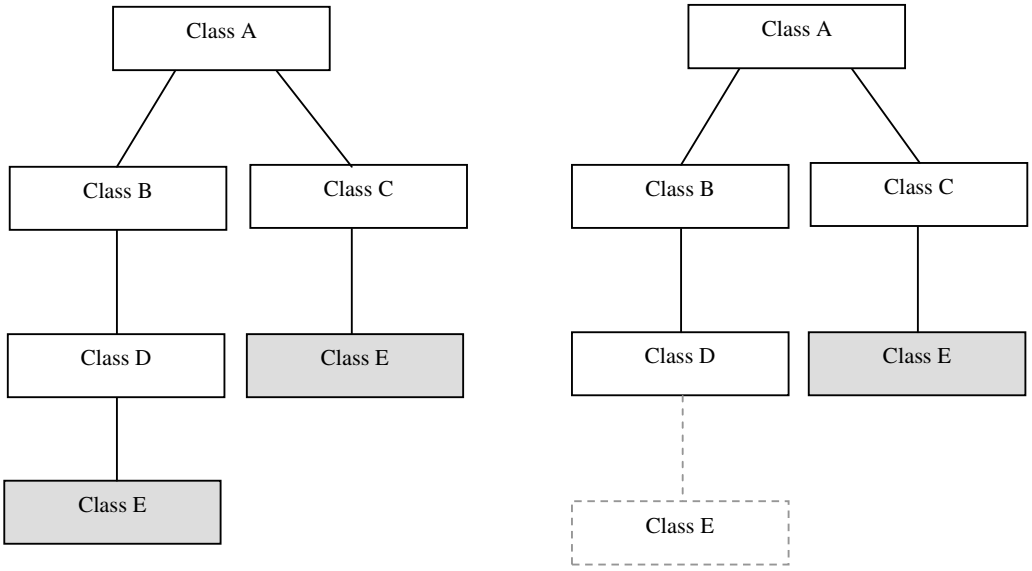


Figure 4-12 Solution adopted to multiple attributes and methods inherited from the same class

Class A inherits twice from class E. In the C++ language, the methods and attributes of each occurrence of class E are distinguished by the class scope operator. For example, from object class A, the separate occurrences of a class E method are identified by B::D::E::method_name() and C::E::method_name() respectively. In the Figure 4-11 entity-relationship model of an object-class, the class scope of inherited elements is not described and so the two occurrences of method_name () are not distinguished. An object-class inheriting from multiple instances of

a single class will be modelled with fewer inherited methods and attributes than it actually has. The mathematical model of the C++ object-class module represents a simplification of the actual, real world object described in the natural language model.

The C++ object-class natural language and mathematical models defined in this section form the core of the following models describing C++ object-class interface dependence, external relationships, connection obscurity and dependency.

4.2.4.2 Interface dependence sub-characteristic of C++ object modularity

In section 3.2.3.1, interface dependence is conceptually defined as the degree to which the module interface elements are dependent on each other and on the implementation specific details of the module. Interface element interdependence, interface size and data exposure are identified as the most refined interface dependence sub-characteristics. The following natural language and mathematical models describe features of C++ software that increase the levels of interface element interdependence, interface size and data exposure present in an object module, thereby reducing its modularity. Natural language model features are followed by their associated mathematical model set name, shown in brackets. A one to one correspondence between natural language feature and mathematical model set indicates that the mathematical model is able to describe the associated feature.

4.2.4.2.1 Natural language model of C++ object interface dependence

5. Interface Dependence : C++ object module
   - 5.1 Interface Element Interdependence - The dependency producing connections between interface elements within the same module.
     - 5.1.1 Object-class (C) interface methods (MM, AM, M) directly read (MCReadA) and/or write (MCWriteA) same object-class (C) interface attributes (MA, AA, A). *Interface methods are directly dependent on interface attributes. Should an interface attribute be modified, there is a chance that interface methods dependent on it will need to be modified too.*
     - 5.1.2 Object-class (C) interface methods (MM, AM, M) indirectly read (MICReadA) and/or write (MICWriteA) same object-class (C) interface attributes (MA, AA, A) *Interface methods are indirectly dependent on interface attributes. Should an interface attribute be modified, there is a chance that interface methods dependent on it will need to be modified too.*

5.1.3 Same object-class (C) interface methods (MM, AM, M) directly invoke (MCInvM) each other.

Interface methods are directly dependent on other interface methods. Should an interface method be modified, there is a chance that other interface methods dependent on it will need to be modified too.

5.1.4 Same object-class (C) interface methods (MM, AM, M) indirectly invoke (MICInvM) each other.

Interface methods are indirectly dependent on other interface methods. Should an interface method be modified, there is a chance that other interface methods dependent on it will need to be modified too.

5.2 Interface Size - The size of a module's interface.

5.2.1 Object-class (C) interface contains attributes (MA, AA, A).

Implementation specific details of the object-class are included in the interface in the form of object-class attributes.

5.2.2 Object-class (C) has a relatively high proportion of total member methods (MM, AM, IAM, M) in the interface.

Methods that are likely to contain implementation specific details of the object-class are included in the interface.

5.2.3 Object-class (C) interface methods (MM, AM, M) are relatively large having:

5.2.3.1 many lines of code (M)

Code that implements the object-class services is likely to be contained within the interface methods.

5.2.3.2 many same object-class (C) method (MM, AM, IAM, M) invocations (MCInvM).

Implementation specific multiple method invocations are contained within the interface methods.

5.2.3.3 many same object-class (C) direct attribute (MA, AA, IAA, A) accesses (MCReadA, MCWriteA).

Implementation specific direct attribute accesses are contained within the interface methods.

5.3 Data Exposure - The degree to which the module data is revealed to the module interface.

5.3.1 Object-class (C) has attributes (MA, AA, A) in the interface.

Object-class attributes, highly implementation specific elements of the object-class, occur in the interface.

5.3.2 Object-class (C) individual interface methods (MM, AM, M) directly read (MCReadA) and/or write (MCWriteA) same object-class (C) attributes (MA, AA, A).

The more attributes an interface methods accesses, the greater the chance that it will be affected by an attribute change.

5.3.3 Object-class (C) individual member attributes (MA, AA, IAA, A) directly read (MCReadA) and/or written (MCWriteA) by same object-class (C) interface methods (MM, AM, M).

The more interface methods directly access an object-class attribute, the greater that chance that a change to that attribute will affected interface method operation.

4.2.4.2.2 Mathematical model of C++ object interface dependence

The entity-relationship model of C++ object interface dependence is illustrated in Figure 4-13. The set definitions of the entities and relationships of this interface dependence model are detailed in Appendix 1. As discussed in section 4.2.4.2.3, situations exist in which this mathematical model is unable to fully describe all the features of the object interface dependence natural language model.

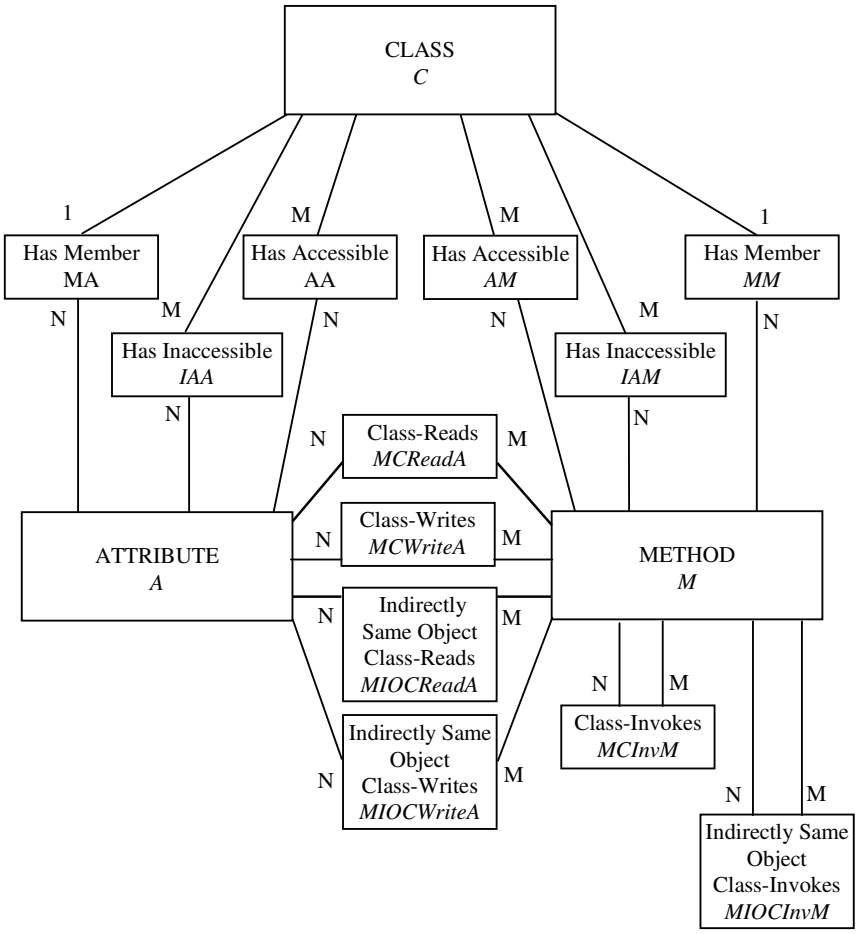


Figure 4-13 C++ object interface dependence mathematical model

4.2.4.2.3 C++ object interface dependence mathematical model completeness

The entity-relationship model of C++ object-class interface dependence captures the natural language model except when the object-class inherits elements from more than one version of an ancestor class. This situation has been previously discussed in the C++ object model description section 4.2.4.1.3. This shortcoming affects all points of the object interface dependence natural language model. Measures of object interface dependence using sets AM, IAM, AA and IAA may have reduced validity for object-classes inheriting multiple times from the same ancestor class.

4.2.4.3 External relationships sub-characteristic of C++ object modularity

In section 3.2.3.2, external relationships is conceptually defined as the degree to which a module has relationships with elements external to itself. External relationships within the system is identified as the most refined external relationships sub-characteristic. The following natural language and mathematical models describe features of C++ software that increase the levels of external relationships of an object module, thereby reducing its modularity.

4.2.4.3.1 Natural language model of C++ object external relationships

6. External Relationships : C++ object module
   - 6.1 Within the System - The relationships between a module and elements external to the module but still within the given software system.
     - 6.1.1 Object-class (C) has one or more immediate supplier objects (MO, AO, IAO, O).
       The object has an association relationship with one or more immediate supplier objects.
     - 6.1.2 Object-class (C) has one or more global supplier objects (O) within its scope (SO).
       The object has an informal association relationship with global supplier objects within its scope.
     - 6.1.3 Object-class (C) instantiates (OIC) one or more objects (O) that are immediate supplier objects (MO, AO, IAO) to a class (C).
       The object has an association relationship with an immediate client class.
     - 6.1.4 Object-class (C) instantiates (OIC) one or more objects (O) that are immediate supplier objects (FIMO) to a global function (F).
       The object has an association relationship with an immediate client global function.

6.1.5 Object-class (C) has a member (MA, AA, IAA) static attribute (A).
A static attribute member of a class forms a connection between all objects instantiated from that class including objects instantiated from classes inheriting from the class with the static member attribute.

6.1.6 Object-class (C) has one or more friend (CEF, CIF) classes (C).
Objects instantiated from a class that has immediate or inherited friend classes can have both hidden and interface elements directly accessed by objects instantiated from the friend classes.

6.1.7 Object-class (C) has one or more friend (FF, IF) global functions (F).
Objects instantiated from a class that has immediate or inherited friend global functions can have both hidden and interface elements directly accessed by the friend global functions.

6.1.8 Object-class (C) is friend (CEF, CIF) to one or more classes (C).
Objects instantiated from a class that is an immediate or inherited friend of another class can directly access both interface and hidden elements of objects instantiated from the other class.

4.2.4.3.2 Mathematical model of C++ object external relationships

The entity-relationship model of C++ object external relationships is illustrated in Figure 4-14. The set definitions of the entities and relationships of this external relationships model are detailed in Appendix 1. As discussed in section 4.2.4.3.3, situations exist in which this mathematical model is unable to fully describe all the features of the object external relationships natural language model.

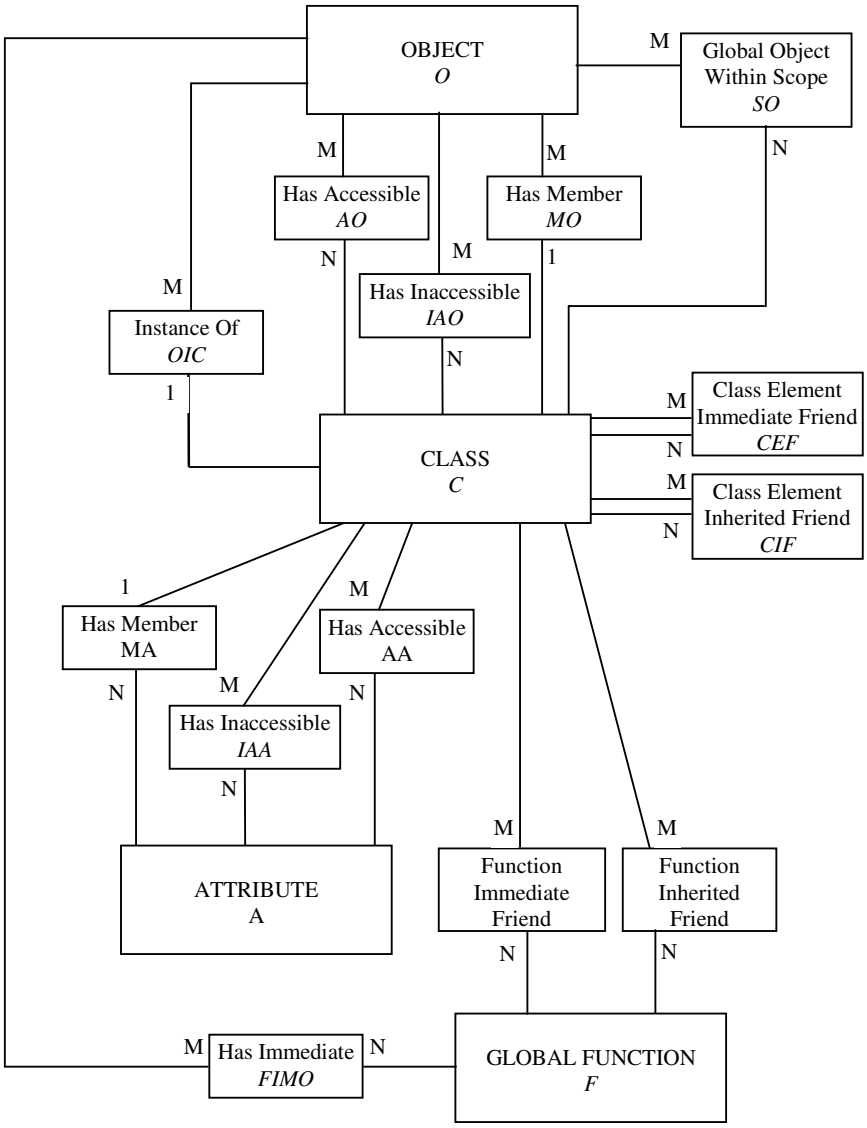


Figure 4-14 C++ object external relationships mathematical model

4.2.4.3.3 C++ object external relationships mathematical model completeness

The entity-relationship model of C++ object-class external relationships captures the natural language model except when the object-class inherits elements from more than one version of an ancestor class. This situation has been previously discussed in the C++ object model description section 4.2.4.1.3. This shortcoming affects points 6.1.1 and 6.1.5 of the object external relationships natural language model. Measures of object external relationships using

sets AM, IAM, AA, IAA, AO and IAO may have reduced validity for object-classes inheriting multiple times from the same ancestor class.

4.2.4.4 Connection obscurity sub-characteristic of C++ object modularity

In section 3.2.3.3, connection obscurity is conceptually defined as the degree to which the connection from a module to an external element is unclear after examination of the module or the external element or both. Variable connection, unstated relationship, distant connection, unexpected relationship and connection via non-standard interface are identified as the most refined connection obscurity sub-characteristics. The following natural language and mathematical models describe features of C++ software that increase the levels of variable connection, unstated relationship, distant connection, unexpected relationship and connection via non-standard interface of an object module, thereby reducing its modularity.

4.2.4.4.1 Natural language model of C++ object connection obscurity

7. Connection Obscurity : C++ object module
   - 7.1 Variable Connection - The nature of the connection from a module to external elements is not fixed.
     - 7.1.1 Object-class (C) has member (MO, AO, IAO) objects (O) declared as a pointer rather than a specific object instance.
       An object declared as a pointer can be referencing none, one or multiple instances of the object. The number of objects referenced by the pointer may change dynamically as the program executes.
     - 7.1.2 Object-class (C) has global objects (O) are declared as a pointer rather than an actual instance within its scope (SO).
       An object declared as a pointer can be referencing none, one or multiple instances of the object. The number of objects referenced by the pointer may change dynamically as the program executes.

7.2 Unstated Relationships - The connection from a module to an external element is not explicitly stated within the module or the external element or both.

7.2.1 Object-class (C) member methods (MM, AM, IAM, M) directly access (MGWriteV, MGReadV) global variables (V).

The relationship between an object whose member, accessible or inaccessible methods directly read or write a global variable can only be discerned from examination of the code that implements the instantiating class and the code of any ancestor classes it may have. This relationship cannot be discerned from examination of the instantiating class declaration or the global variable declaration.

7.2.2 Object-class (C) member methods (MM, AM, IAM, M) directly invoke (MGInvF) global functions (F).

In a similar manner to global variable access, this relationship cannot be discerned from examination of the instantiating class declaration or the global function definition.

7.2.3 Object-class (C) member methods (MM, AM, IAM, M) directly access (MOAccessO) global objects (O).

The informal association relationship between an object whose member, accessible or inaccessible methods directly access a global object can only be discerned from examination of the code that implements the accessing object's instantiating class and the code of any ancestor classes it may have. This relationship cannot be discerned from examination of the instantiating class declaration or the global object's instantiating class declaration.

7.2.4 Object-class (C) has accessible or inaccessible (AA, IAA) static attributes (A).

The inheritance of a static attribute provides an unstated relationship between the set of objects comprised of all objects instantiated from the parent class within which the static attribute is declared and all objects instantiated from descendent classes of this parent.

7.2.5 Object-class (C) has inherited (IP, IDA) friend (CEF) classes (C).

A friend type relationship is inherited. A friend class can directly access all the elements a class inherits from an ancestor with that friend class. The friend class may be able to directly access some or all of the accessible and inaccessible elements of objects instantiated from the descendent class. This results in an unstated relationship between the descendent classes and the friend class.

7.2.6 Object-class (C) has inherited (IP, IDA) friend (FF) global functions (F).
An immediate or distant inheritance relationship with a class having a declared friend global function results in an unstated relationship between the descendent classes and the friend global function. This is because the friend global function may be able to directly access some or all of the accessible and inaccessible elements of the descendent class.

7.3 Distant Connection - The connection is between a module and an external element with a distant, indirect relationship.

7.3.1 Object-class (C) member methods (MM, AM, IAM, M) directly access (MGReadV, MGWriteV) a global variable (V) that is also directly accessed (MGReadV, MGWriteV) by member methods (MM, AM, IAM, M) of another object-class (C).
An object has an indirect, distant connection to another object when they share access to a common global variable.

7.3.2 Object-class (C) member methods (MM, AM, IAM, M) directly access a non-global object (O) to which the object-class is not directly associated (**no model set defined**).
An object has an indirect, distant connection to another non-global object when it accesses elements of the object even thought it does not have a direct association relationship with the object. This situation occurs when an object associated with an object-class has another object visible in its interface. The object-class can directly access this distant interface object.

7.4 Unexpected Relationship - The relationship from a module to an external element is an unexpected one that users would not commonly look for and recognise.

7.4.1 Object-class (C) member methods (MM, AM, IAM, M) directly access (MGWriteV, MGReadV) one or more global variables (V).
Connection between objects of the same class type via a shared global variable is not a standard relationship between objects.

7.4.2 Object-class (C) member methods (MM, AM, IAM, M) directly access (MGReadV, MGWriteV) a global variable (V) that is also directly accessed (MGReadV, MGWriteV) by member methods (MM, AM, IAM, M) of another object-class (C).
Connection between objects of different class type via a shared global variable is not a standard relationship between objects.

7.4.3 Object-class (C) has one or more friend (CEF, CIF) classes (C).
Friendship is not the standard relationship between objects.

7.4.4 Object-class (C) has one or more friend (FF, FIF) global functions (F).
Friendship is not the standard relationship between global functions and objects.

7.4.5 Object-class (C) is friend (CEF, CIF) to one or more classes (C).
Friendship is not the standard relationship between objects.

7.4.6 Object-class (C) has one or more member (MA, AA, IAA) static attributes (A).
Connection between objects via a shared static attribute is not a standard relationship between objects.

7.5 Connection via Non-standard Interface - Connection between a module and an external element takes place outside the standard module interface.

7.5.1 Object-class (C) member methods (MM, AM, IAM, M) directly access (MGReadV, MGWriteV) a global variable (V) that is also directly accessed (MGReadV, MGWriteV) by member methods (MM, AM, IAM, M) of another object-class (C).
A global variable is not part of the standard interface of an object.

7.5.2 Object-class (C) member methods (MM, AM, IAM, M) directly access the attributes (MOAccessO) of an object (O).
A standard object interface does not contain attributes.

7.5.3 Object-class (C) member methods (MM, AM, IAM, M) directly access the hidden elements (**no model set defined**) of an object (O).
An object's hidden elements are not part of its standard interface.

7.5.4 Object-class (C) member methods (MM, AM, IAM) directly access the inaccessible elements (**no model set defined**) of an object (O).
An object's inaccessible elements are not part of its standard interface.

7.5.5 Object-class (C) attributes directly accessed (MOAccessO, OIC) by the member methods (MM, AM, IAM, M) of another object-class (C).
A standard object interface does not contain data elements.

7.5.6 Object-class (C) attributes directly accessed (FOAccessO, OIC) by a global function (F).
A standard object interface does not contain data elements.

7.5.7 Object-class (C) hidden elements directly accessed (**no model set defined**) by member methods (MM, AM, IAM, M) of another object-class (C).
Hidden elements are not part of an object's interface.

7.5.8 Object-class (C) inaccessible elements directly accessed (**no model set defined**) by member methods (MM, AM, IAM, M) of another object-class (C).
Inaccessible elements are not part of an object's interface and are in fact inaccessible to their own object's member elements thus for an external object to be able to access them is a highly non-standard form of connection.

7.5.9 Object-class (C) hidden elements directly accessed (**no model set defined**) by a global function (F).
Hidden elements are not part of an object's interface.

7.5.10 Object-class (C) inaccessible elements directly accessed (**no model set defined**) by a global function (F).
Inaccessible elements are not part of an object's interface and are in fact inaccessible to their own object's member elements thus for a global function to be able to access them is a highly non-standard form of connection.

4.2.4.4.2 Mathematical model of C++ object connection obscurity

The entity-relationship model of C++ object connection obscurity is illustrated in Figure 4-15. The set definitions of the entities and relationships of this connection obscurity model are detailed in Appendix 1. As discussed in section 4.2.4.4.3, situations exist in which this mathematical model is unable to fully describe all the features of the object connection obscurity natural language model.

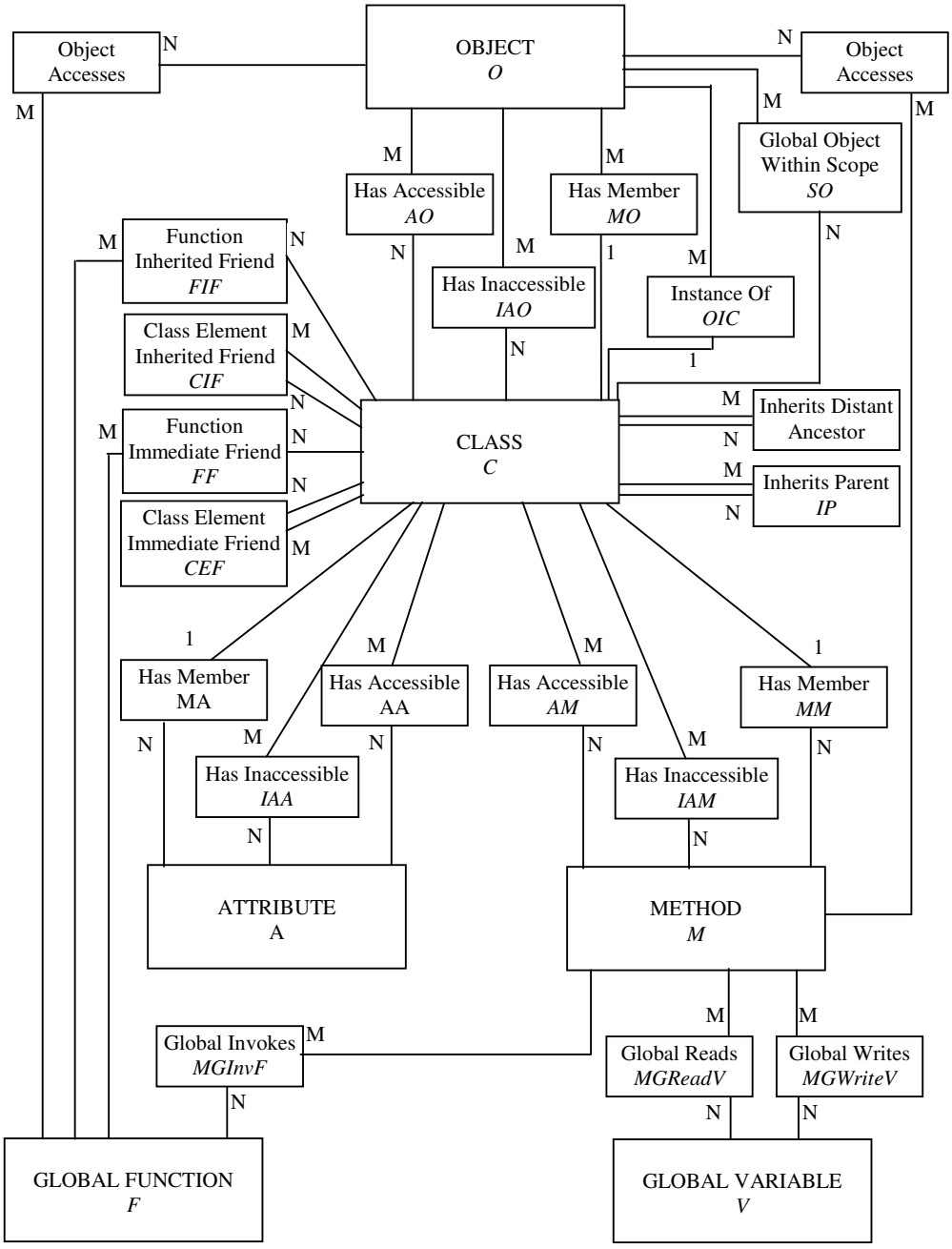


Figure 4-15 C++ object connection obscurity mathematical model

4.2.4.4.3 C++ object connection obscurity mathematical model completeness

The entity-relationship model of C++ object-class connection obscurity is unable to completely describe all the elements of an object-class when it inherits elements from more than one version of an ancestor class. This situation has been previously discussed in the C++ object model description section 4.2.4.1.3. This shortcoming affects points 7.1.1, 7.2.4, 7.2.5, 7.2.6 and 7.4.6 of the object connection obscurity natural language model. Measure of object connection obscurity using sets AM, IAM, AA, IAA, AO and IAO may have reduced validity for object-classes inheriting multiple times from the same ancestor class.

Another shortcoming of the object connection obscurity mathematical model is that it is unable to describe the particular object methods or attributes accessed by a method or global function. This is because each object entity in the model is not assigned its own instances of methods and attributes. The mathematical model is only able to describe, in relations MOAccessO and FOAccessO, the more general information that an object is accessed by a method or global function and that this access is either an attribute read, attribute write or method invocation.

$$MOAccessO = \{(mi, oi, action)\} = \{(m.mi, o.oi, action) \mid m \in M \wedge o \in O \wedge action \in \{read, write, invoke) \wedge (method\ m\ directly\ object\text{-}accesses\ object\ o)\}$$

$$FOAccessO = \{(fi, oi, action)\} = \{(f.fi, o.oi, action) \mid f \in F \wedge o \in O \wedge action \in \{read, write, invoke) \wedge (global\ function\ f\ directly\ object\text{-}accesses\ object\ o)\}$$

The implication of this is that object connection obscurity natural language model points 7.5.3, 7.5.4, 7.5.7, 7.5.8, 7.5.9 and 7.5.10 cannot be described by measures defined in terms of the object connection obscurity mathematical model. The aspects of object connection obscurity described by points 7.5.3, 7.5.4, 7.5.7, 7.5.8, 7.5.9 and 7.5.10 only arise when an object has a friend relationship with a class or global function. The connection obscurity measurement data obtained from objects in a friend relationship should be interpreted in the knowledge that the measured values possibly underestimate the level of connection obscurity present in these objects.

4.2.4.5 Dependency sub-characteristic of C++ object modularity

In section 3.2.3.4, dependency is conceptually defined as the degree to which a module depends on external elements in order that it can perform its own functions correctly. Service invocation, interface provision, external variable reading and external function writing are identified as the most refined dependency sub-characteristics. The following natural language and mathematical models describe features of C++ software that increase the levels of service invocation, interface provision, external variable reading and external function writing of an object module, thereby reducing its modularity.

4.2.4.5.1 Natural language model of C++ object dependency

8. Dependency : C++ object module
   - 8.1 Service Invocation - The dependence of a module on services provided by elements external to itself.
     - 8.1.1 Object-class (C) member methods (MM, AM, IAM, M) directly invoke (MGInvF) one or more global functions (F).
       To perform its own tasks, an object needs to invoke the services of a global function.
     - 8.1.2 Object-class (C) member methods (MM, AM, IAM, M) directly invoke methods (MOAccessO) of other objects (O).
       To perform its own tasks, an object needs to invoke the services of another object.
   - 8.2 Interface Provision - The dependency of a module on external modules to provide some or all of its interface elements.
     - 8.2.1 Object-class (C) interface elements (AM, M, AA, A) provided by ancestor classes.
       The class from which it is instantiated does not directly provide all the interface elements of an object. Inherited elements appear within the object's interface.
   - 8.3 External Variable Reading - The degree to which the module's state is dependent on values contained in external variables.
     - 8.3.1 Object-class (C) member methods (MM, AM, IAM, M) directly read (MGReadV) global variables (V).
       The object relies on state information held in a global variable that is external to itself.
     - 8.3.2 Object-class (C) member methods (MM, AM, IAM, M) directly read an attribute (MOAccessO) of another object (O).
       The object relies on state information held within an object external to itself.

8.4 External Function Writing - The degree to which the module's state is dependent on values written, by external functions, to attributes or external variables from which the module directly reads.

8.4.1 Object-class (C) member methods (MM, AM, IAM, M) directly read (MGReadV) from a global variable (V) directly written (MGWriteV) by member methods (MM, AM, IAM, M) of another object-class (C).
The object relies on another object to maintain a global variable with a value compatible with its own correct operation.

8.4.2 Object-class (C) member methods (MM, AM, IAM, M) directly read (MGReadV) from a global variable (V) directly written (FGWriteV) by a global function (F).
The object relies on a global function to maintain a global variable with a value compatible with its own correct operation.

8.4.3 Object-class (C) attribute directly written (FOAccessO) by a global function (F).
The object relies on a global function to maintain one of its own attributes with a value compatible with its correct operation.

8.4.4 Object-class (C) member methods (MM, AM, IAM, M) directly class-read (MCReadA) from a static attribute (A) that is directly class-written (MCWriteA) by a member method (MM, AM, IAM, M) of another object class (M).
The object relies on another object to maintain a static attribute with a value compatible with its own correct operation.

8.4.5 Object-class (C) member methods (MM, AM, IAM, M) directly object-read a static attribute (**not described by model**) directly object-written to (**no model set defined**) by another object-class (C).
The object relies on another object to maintain a static attribute with a value compatible with its own correct operation.

8.4.6 Object-class (C) member attributes (MA, AA, IAA, A) directly object-written to (MOAccessO) by the member methods (MM, AM, IAM, M) of another object-class (C).
The object relies on another object to maintain an attribute with a value compatible with its own correct operation.

4.2.4.5.2 Mathematical model of C++ object dependency

The entity-relationship model of C++ object dependency is illustrated in Figure 4-16. The set definitions of the entities and relationships of this connection obscurity model are detailed in

Appendix 1. As discussed in section 4.2.4.5.3, situations exist in which this mathematical model is unable to fully describe all the features of the object dependency natural language model.

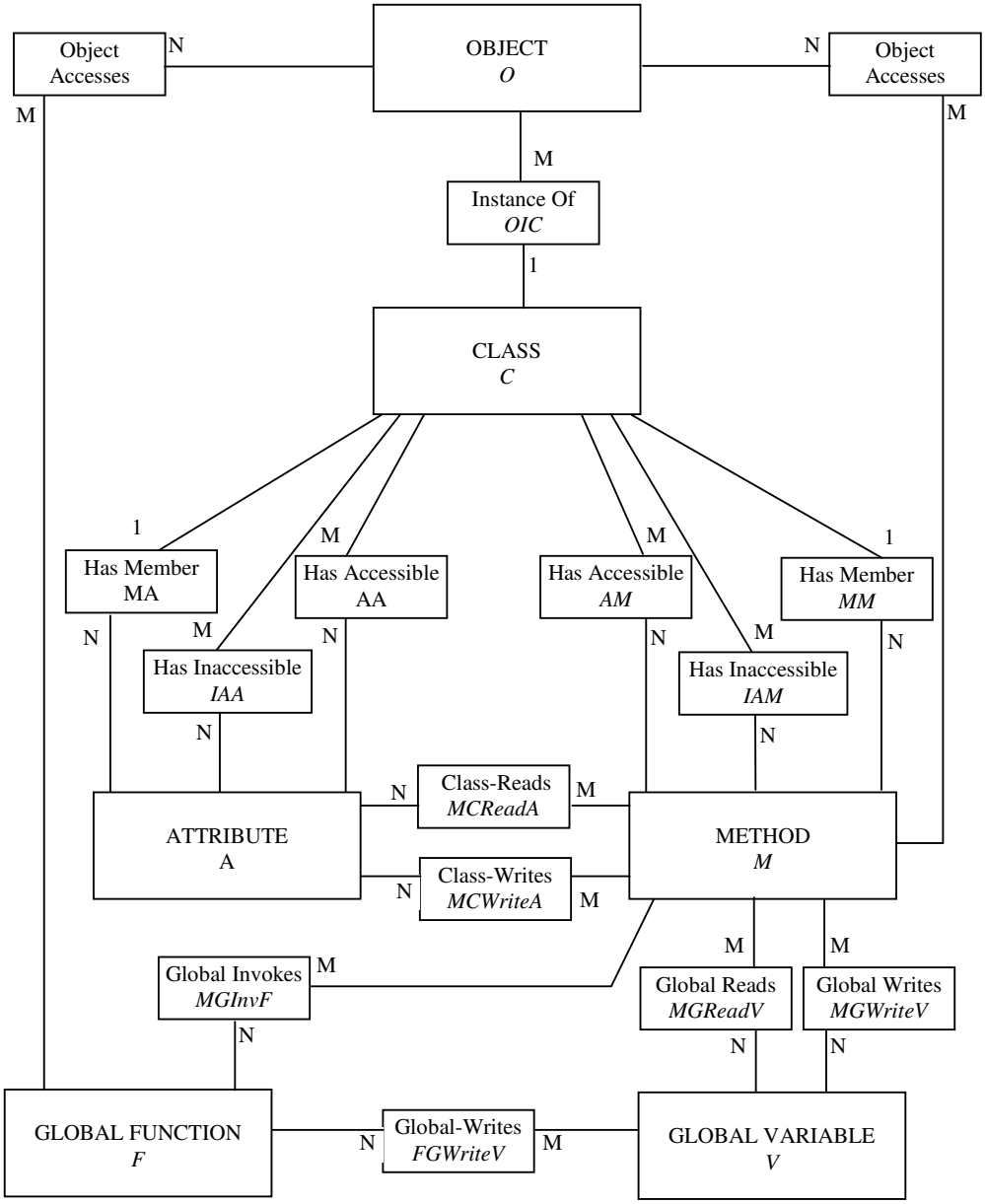


Figure 4-16 C++ object dependency mathematical model

4.2.4.5.3 C++ object dependency mathematical model completeness

The entity-relationship model of C++ object-class dependency is unable to completely describe all the elements of an object-class when it inherits elements from more than one version of an ancestor class. This situation has been previously discussed in the C++ object model description section 4.2.4.1.3. This shortcoming affects point 8.2.1 of the object external relationships natural language model. Measure of object dependency using sets AM, IAM, AA and IAA may have reduced validity for object-classes inheriting multiple times from the same ancestor class.

Another shortcoming of the object dependency mathematical model is that it is unable to describe the particular object methods or attributes accessed by a method or global function. This situation has been previously discussed in the C++ object connection obscurity mathematical model completeness section 4.2.4.4.3. The implication of this is that object dependency natural language model point 8.4.5 cannot be described by measures defined in terms of the object connection obscurity mathematical model. This aspect of object dependency only arises when one or more object-classes have static attributes. The dependency measurement data obtained from software systems containing static attributes should be interpreted in the knowledge that the measured values possibly underestimate the level of dependency present in some objects.

4.2.4.6 Summary: object mathematical model completeness

In certain situations, the C++ object mathematical entity-relationship models are unable to fully describe the object modularity natural language models. These shortcomings have been discussed in sections 4.2.4.1.3, 4.2.4.2.3, 4.2.4.3.3, 4.2.4.4.3 and 4.2.4.5.3. If a software system contains features that cannot be modelled by the object mathematical models, then the system description obtained from these models will be incomplete.

Table 4-3 summarises the shortcomings of the C++ object modularity entity-relationship mathematical models. Each shortcoming is assigned an identification number, described, and the situation in which the mathematical model is unable to fully describe the software identified.
The final column of Table 4-3 indicates measures that can be used to identify object-classes that are possibly incompletely described by object mathematical models. Object-classes

identified in this way should be further examined to determine whether this is the case. Measures CER2, OER5:OCM4, OER6:OCM5, OER7:OCM6 and OER8:OCM7 are defined in Chapter 5.

Id.	Mathematical model shortcoming	Situation in which it occurs	Indicative measure
1	Unable to describe all object inherited attributes and methods.	Inheritance of more than one copy of the same class.	Only possible in classes with one or more distant ancestor classes. CER2.total > 0
2	Unable to record that object hidden or inaccessible elements are directly accessed by an external method or global function.	When an object-class with hidden or inaccessible elements has immediate or inherited friend classes or global functions.	Only possible when class or global function are friend to the accessed object-class. OER6:OCM5.total > 0 OER7:OCM6.total > 0
3	Unable to record that object-class directly accesses the hidden or inaccessible elements of another object.	Where an object-class is an immediate or inherited friend to objects with hidden or inaccessible elements.	Only possible when object-class is friend to another object-class. OER8:OCM7.total > 0
4	Unable to record that an object directly reads or writes another object's static attribute.	Where an object-class accesses an object with a static attribute.	Only possible in a software system with static attributes. OER5:OCM4.total > 0

Table 4-3 Shortcomings of C++ object modularity mathematical models

The following points indicate the particular mathematical model shortcomings that affect each object modularity model.

- Object entity model has shortcoming 1.
- Interface dependence object model has
  - shortcoming 1 affecting all points of the natural language model.
- External relationships object model has
  - shortcoming 1 affecting points 6.1.1 and 6.1.5 of the natural language model.

- Connection obscurity object model has
  - shortcoming 1 affecting points 7.1.1, 7.2.4, 7.2.5, 7.2.6 and 7.4.6 of the natural language model.
  - shortcoming 2 affecting points 7.5.7, 7.5.8, 7.5.9 and 7.5.10 of the natural language model.
  - shortcoming 3 affecting points 7.5.3 and 7.5.4 of the natural language model.
- Dependency object model has
  - shortcoming 1 affecting point 8.2.1 of the natural language model.
  - shortcoming 4 affecting point 8.4.5 of the natural language model.

Once those object-classes possibly affected by the described situation are identified using the indicative measures, they can be examined using code analysis software such as Understand for C++ (Scientific Toolworks Inc. 2004) to determine whether or not the identified shortcoming applies. If it does, then the object-class will not be completely described by the mathematical model. This in turn means that some measures describing the modularity of this object cannot be fully implemented. In this situation, a potential user must decide whether or not to proceed with measuring the software system given that the modularity of some objects will not be fully described by the measures. If the user does proceed with the measurement, data analysis may need to take into account the fact that the modularity of the affected object classes may be less than the measured values would indicate. The issue of measure validity is discussed in greater detail in Chapter 5.

4.2.4.7 Object modularity sub-characteristics sharing identical feature descriptions

Similarly to class modularity, several modularity sub-characteristics are listed in the natural language model of C++ object modularity with the same features affecting their presence in the software. The implications of this are the same as those for class modularity measurement, as discussed in Section 4.2.3.6. Table 4-4 lists these sub-characteristics with their associated common features. Table 5-32 in Chapter 5 lists the measures quantifying common software features of C++ object modularity.

Object Modularity Sub-characteristics	Feature
Interface Dependence : Interface Implementation Dependence: Interface Size Interface Dependence : Interface Implementation Dependence: Data Exposure	5.2.1 5.3.1
Interface Dependence : Interface Implementation Dependence: Interface Size Interface Dependence : Interface Implementation Dependence: Data Exposure	5.2.3.3 5.3.2
External Relationships Connection Obscurity : Non-standard Connection : Unexpected Relationship	6.1.5 7.4.6
External Relationships Connection Obscurity : Non-standard Connection : Unexpected Relationship	6.1.6 7.4.3
External Relationships Connection Obscurity : Non-standard Connection : Unexpected Relationship	6.1.7 7.4.4
External Relationships Connection Obscurity : Non-standard Connection : Unexpected Relationship	6.1.8 7.4.5
Connection Obscurity : Unstated Relationship Connection Obscurity : Non-standard Connection : Unexpected Relationship	7.2.1 7.4.1
Connection Obscurity : Unstated Relationship Dependency : Service Invocation	7.2.2 8.1.1
Connection Obscurity : Distant Connection Connection Obscurity : Non-standard Connection : Unexpected relationship Connection Obscurity : Non-standard Connection : Connection via Non-standard Interface	7.3.1 7.4.2 7.5.1

Table 4-4 Object modularity sub-characteristics sharing common natural language model features

4.3. Conclusion

Entity modelling to describe the features that affect the levels of a characteristic present in an entity is the second stage of the systematic measure development process. A natural language entity model is an essential product of this stage. An accompanying mathematical model is optional but advantageous when precise, unambiguous measure definitions can be made in terms of this mathematical model. Prerequisite to entity modelling is the conceptual definition of the characteristic to be described by the developed measures, and a theoretical basis describing the ways in which the features of the entity to be measured affect the levels of characteristic present. Together, the conceptual definitions and natural language entity models express the theoretical basis from which the descriptive measures are developed. The natural language entity model is an essential, and the mathematical model an optional prerequisite to the subsequent measure operational definition stage of descriptive measure development.

The natural language entity model is important because it specifically describes the links from the tangible features of the entity, quantified by the descriptive measures, to the entity characteristic of interest. This information facilitates the appropriate application of the measures as well as the subsequent analysis and interpretation of measured data. The mathematical model is important because it precisely and unambiguously describes the software and provides a framework within which measures can be concisely and unambiguously defined in terms of the mathematical model components. It is important that the mathematical model be used in conjunction with a natural language model because only the natural language model describes how the software features described by the mathematical model affect the levels of characteristic of interest present in the software. When deciding to use a mathematical entity model, consideration should be given to the validity of its description of the software. Ideally, the selected type of mathematical model is able to describe all the software features of the natural language model. In practice, this may not be the case and the measure development process may need to accommodate some model shortcomings or use alternative mathematical models.

In this chapter, the process of entity modelling of C++ software has been demonstrated. The conceptual definitions of modularity, described in section 3.2, along with Meyer's (1997) rules of object oriented software modularity, provided the basis for this entity modelling. Entity-relationship type mathematical models describe the software features identified in the natural language model as affecting the levels of class and object modularity present in the C++ software. Some features of this natural language model are only partially described by their associated mathematical model and some features are not described at all. These shortcomings are documented and should be considered at the operational definition and measurement instrument implementation stages of measure development. They will also need to be considered whenever the developed measures are applied, analysed and interpreted.

The operational measure definition stage of measure development, as presented in Chapter 5, can proceed once the prerequisite natural language and optionally, mathematical models, have been defined to describe the features of the software that affect the levels of characteristic present.

5. Operational Measure Definition

This chapter describes the operational definition stage of software descriptive measure development. This stage corresponds to the shaded box in the Figure 5-1 diagrammatic representation of the measure development process. Section 5.1 of this chapter discusses the development of measures to operationally define a characteristic. Section 5.2 demonstrates operational definition by developing operational measure definitions of C++ class and object modularity, based on the C++ class and object modularity natural language and mathematical entity models developed in section 4.2 of Chapter 4.

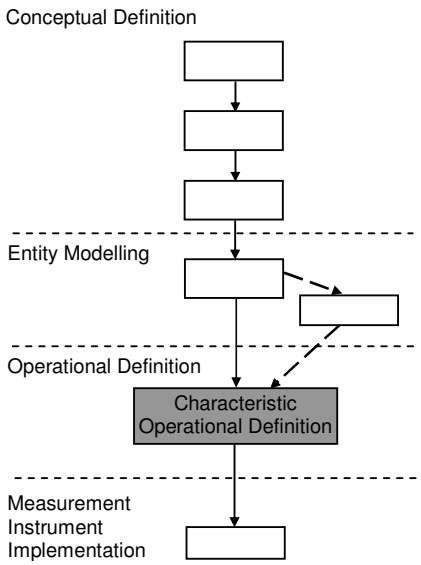


Figure 5-1 Operational definition stage of the measure development process

5.1. Stage 3 of measure development process - operational definition

In general terms, "An operational definition … describes the meaning of a concept through specifying the procedures or operations necessary to measure it." (Diamantopoulos & Schlegelmilch 1997, p. 22). When developing descriptive measures of software characteristics, the operational definition stage describes the software characteristic of interest in terms of natural language measure definitions and optional mathematical measure definitions. These measures quantify the features of the software identified in the natural language entity models as affecting the levels of characteristic of interest present in the software. The natural language entity models identify these features and describe the ways in which they affect the levels of characteristics and sub-characteristics present in the software. At least one measure should be defined to describe each feature of the natural language entity model. If a natural language model feature is not described by any measures, then the final description obtained from the full set of measure will be lacking in that particular detail.

Natural language measure definitions are initially more easily understood by a user; however it can be difficult to precisely define a complex measure in this way. A mathematical measure definition can supplement the natural language definition by providing a precise, unambiguous measure definition. Ideally, natural language and mathematical measure definitions are developed to impart both readability and precision to the operational definition stage of measure development. Both natural language and mathematical measure definitions have been used by previous software measure development projects to describe software characteristics. Chidamber and Kemerer (1994) define measures of object oriented class design complexity using a combination of natural language and some mathematical measure definitions. Briand, Daly and Wust (1997b; 1999a) redefine some of these measures less ambiguously in terms of a more precise mathematical model of the software. Abreu and Carapuca (1994) mathematically define measures of object oriented software quality based on a mathematical software model. Li (1998) defines measures only in natural language terms. These measure development projects show that both natural language and mathematical measure definitions are appropriate for the development of measures that quantify software structural features.

Specifying the particular aspect of the characteristic of interest described by each measure is as important as clearly and unambiguously defining the measure because this information is needed to select appropriate measures for a particular application and to interpret the measured data. The operational measure definition stage of the systematic process of measure development described in this thesis directly links each measure definition to the particular aspect of the characteristic of interest it describes. These links are established via the natural language entity model. The clear and explicit statement of the links from measures to natural language model features to characteristics differentiates the measures developed in this thesis from other previously developed sets of object oriented software measures. For example, Chidamber and Kemerer (1994, p. 483) define a measure of depth of inheritance tree (DIT). This measure quantifies an inheritance feature of object oriented software, and the "viewpoints" (Chidamber and Kemerer 1994, p.483) that accompany this measure definition directly relate DIT to class complexity. What is missing is a description of the particular aspect of class complexity affected by the depth of the class in the inheritance tree. An entity model of class complexity, similar to that developed in Chapter 4 could have provided this information. As it is, a user of the DIT measure must interpret measured data based on the general statement that "The deeper a class is in the [inheritance] hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behaviour." (Chidamber & Kemerer 1994, p. 483).

The process of measure development described and demonstrated in this thesis supports the development of measures based on explicit descriptions of the ways in which each measured software feature affects the levels of characteristics and sub-characteristics of interest present in the software. In this thesis, the operational definition stage results in measure definitions with accompanying information identifying the particular aspect of a characteristic that each measure describes. The following sections describe the operational definition stage of measure development in terms of its prerequisites, performance and products.

5.1.1 Prerequisites to operational definition stage

The essential prerequisites to the operational definition stage are the conceptual definition of the characteristic to be described by the measures and a natural language entity model describing software features that affect the level of this characteristic present in software. An optional but useful prerequisite is a mathematical model of the software describing these identified software features.

5.1.2 Performance of operational definition stage

Once the prerequisite requirements have been met, the operational measures can be defined. One or more measures should be defined to quantify each feature identified in the software characteristic natural language model. Measures should be defined in natural language terms and, where a suitable mathematical model has been developed, the measure may also be more precisely defined in terms of this mathematical model.

5.1.3 Product of operational definition stage

The product of the operational definition stage is a set of measures. These measures quantify the software features that are identified in the natural language entity model and which affect the level of characteristic of interest present in the software. Each measure should be directly associated with a feature of the natural language model so that it can be traced through this model to the particular aspect of the characteristic of interest it describes. This traceability will facilitate measure content validation and support a user as they analyse and interpret the data obtained from applying the measures to a software system. The **char**acteristic to **me**asure **r**elationship (CHARMER) diagram of Figure 5-2 describes the relationships between measures, the natural language entity model features they quantify and the software characteristics thus described.

Using the CHARMER diagram, a measure can be traced to the sub-characteristics and characteristics it describes, and conversely, a characteristic can be traced to the measures that describe it.

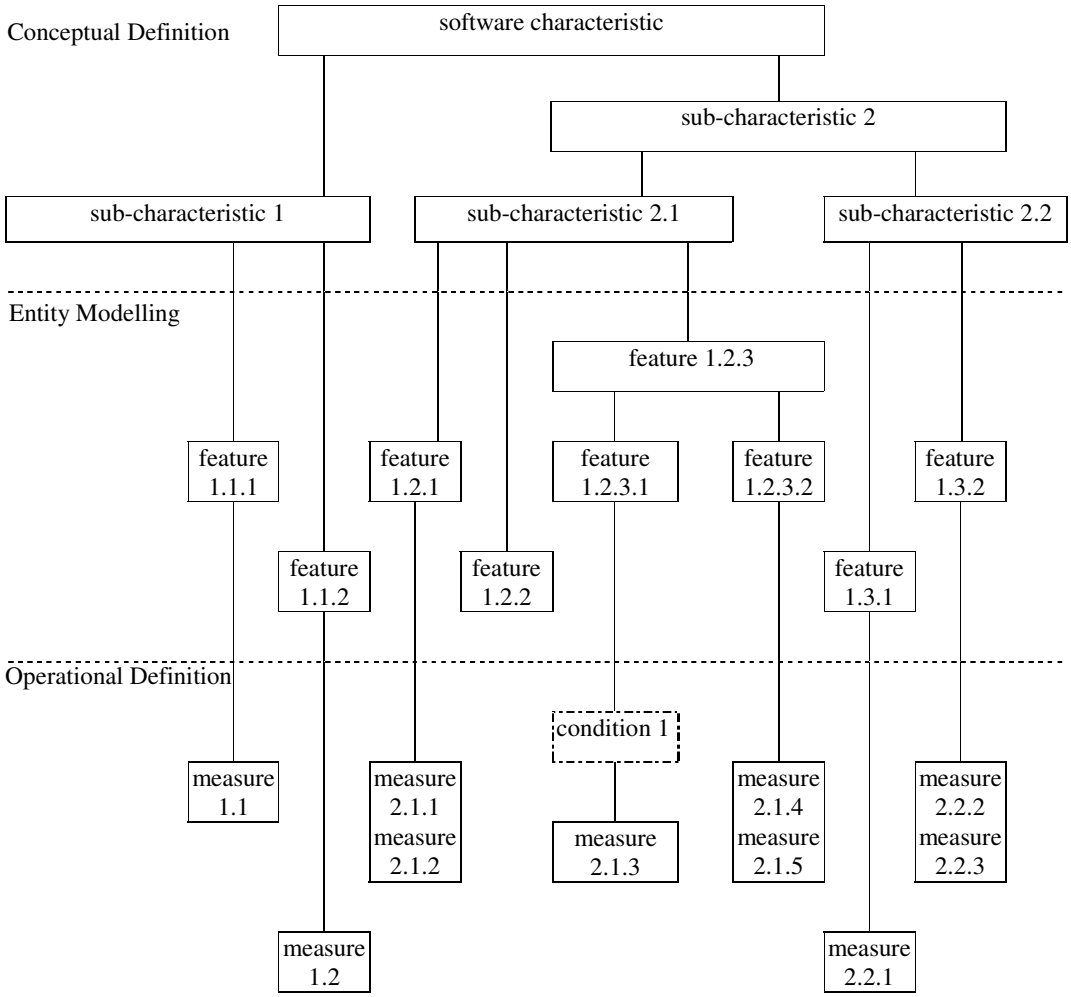


Figure 5-2 Characteristic to measure relationship (CHARMER) diagram

The boxes of the Figure 5-2 CHARMER diagram represent products of the measure development process. Figure 5-2 shows that the conceptual definition stage of measure development can produce a sub-characterisation of the characteristic of interest. The entity modelling stage follows from conceptual definition by identifying software features affecting the levels of characteristic and sub-characteristics present in the software. This in turn leads to the definition of a set of measures to quantify the identified features. The lines connecting the boxes in Figure 5-2 indicate development links between these products. These lines may be

annotated to indicate conditional links between products. For example, in Figure 5-2, measure 2.1.3 provides a description of software feature 1.2.3.1 only if condition 1 is met. A product that does not proceed to the next stage indicates a problem encountered in the measure development process.

The result of any failure to proceed is that an aspect of the characteristic of interest will not be described by a measure. For example, in Figure 5-2, software feature 1.2.2 is not described by a measure. This means that an aspect of sub-characteristic 2.1 is not described and from this, an aspect of sub-characteristic 2 is not described and finally, an aspect of the main software characteristic is not described by a measure. The practical application of CHARMER diagrams to describe the operational definition of a characteristic is demonstrated in the section 5-2 operational definition of C++ class and object modularity measures.

5.1.4 Assessing measure level of measurement and validity

The Figure 1-3 illustration of the systematic descriptive measure development process shows that operational measure definition validity and level of measurement should be assessed. The level of measurement is an individual trait of each measure and describes in general terms the amount of information conveyed by the measure, and the type of "mathematical/statistical operations that can be applied to the resulting data." (Diamantopoulos & Schlegelmilch 1997, p. 24).

Validation demonstrates the degree to which the measures provide an adequate description of the characteristic of interest. While there are several different types of validity, content validity is most appropriate for the operational definition stage of measure development as it is designed to demonstrate the degree to which the defined measures quantify the software features identified in the natural language entity model.

5.1.5 Practical Considerations

If the conceptual definition and entity modelling stage of measure development have been executed with sufficient care and attention to detail, the operational definition stage should proceed smoothly with measures being defined to quantify the software features that have already been identified in the natural language models. If problems are encountered in

identifying the features to quantify or if the measures appear to provide an insufficiently detailed description of the software, then the measure developer may need to repeat the conceptual definition and/or entity modelling stages.

Assuming sufficiently detailed conceptual definitions and entity models, the main operational definition challenges are ensuring that the measure definitions are unambiguous and that they can be understood. A mathematical measure definition aims to reduce the ambiguity of a measure definition, while an accompanying natural language definition will help a potential user understand this definition.

Once the set of measures has been defined, their level of measurement and the validity should be assessed. While determining the level of measurement achieved by a measure is straightforward, assessment of the validity of the set of measures is a more complex task. For the measures developed using the process described in this thesis, demonstration of content validity is appropriate. This does not mean that other forms of validation are not important or relevant. Once content validity has been established, demonstration of other types of validity will mean that the measures can be used for tasks such as predicting future software characteristics or validating other measures describing the same characteristic. Such further validation is beyond the scope of this study but could form the basis of future work.

The following section 5.2 demonstrates the operational definition of C++ class and object modularity by defining descriptive measures to quantify the software features identified in the Chapter 4, section 4.2 natural language models of C++ class and object modularity.

5.2. Operational definition of C++ class and object modularity

This section describes the operational definition of C++ class and object modularity. Features identified in the natural language models of class and object modularity will be quantified by the defined measures. Both natural language and mathematical measure definitions are developed. The natural language definitions are intended to be used only as a guide to reading the mathematical definitions. For this reason, the natural language definitions, while relatively easy to understand, are too general to be used on their own The mathematical definitions offer sufficient precision to operationally define the characteristic but are somewhat more difficult to understand.

The first part of this section discusses the notation used to mathematically define the measures. Following this, the measures describing C++ class and object modularity are fully defined. The links between these measures and the modularity sub-characteristics they describe are detailed in CHARMER diagrams. The final part of this section discusses content validation of the modularity measures. The sub-characteristics to measure relationship (CHARMER) diagrams facilitate this content validation.

The measures are named according to the sub-characteristic they describe. All class modularity measures start with a C and all object modularity measures start with an O. Multiple measures describing the same sub-characteristic are distinguished by number.

5.2.1 Prerequisites

The prerequisite to the measure operational definition stage is an entity model describing the features of the software that affect the levels of characteristic present. The entity models developed in section 4.2 of the previous chapter form the basis of the work performed in this section. A mathematical entity model has been defined and so, measures are mathematically defined in terms of this model. The following section discusses the notation used to define these measures.

5.2.2 Mathematical measure definition notation

The relational calculus notation described by Grassmann and Tremblay (1996, pp. 620-624) is used to mathematically define the modularity measures. This style of mathematical measure definition is based on the style of measure definition employed by Briand, Daly and Wust (1997, 1999) and is similar to that of Arisholm, Briand and Foyen, (2004) although developed independently of these researchers. The advantage of using relational calculus notation is that it readily converts to relational database queries (Grassmann & Tremblay 1996, p. 620), which facilitates the implementation of the measures in a relational database application. The following symbols are used in the measure definitions.

∃	there exists	∧	and
¬∃	there does not exist	∨	or
|	where	∪	union
∈	is an element of	∩	intersection
∉	is not an element of	∀	for all
≠	logical inequality	#	number of elements in the set
=	can mean either logical equality or assignment equality depending on the context		

(a, total)	the element pair of 'a' and 'total'
{(a, total)}	the set of all element pairs of 'a' and 'total'

when y ∈ {(a, total)}, y.a refers to the 'a' part of element y and y.total refers to the 'total' part of element y.

The following example defines measure CIS1 in natural language and mathematical terms using relational calculus notation and demonstrates how the relational calculus notation can be interpreted.

> Measure definition CIS1 quantifies, for each class, the number of member interface attributes.
>
> CIS1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ai | y ∈ A ∧ y.ci = x.ci ∧ y.protection ∈ {public, protected}}}
>
> where
>
> > 'C' denotes the set of classes {(ci, name)} where 'ci' is a unique integer identifier and 'name' is a valid C++ class name.
> >
> > 'A' denotes the set of attributes {(ai, ci, protection, name)} where ('ai' is a unique integer identifier) and 'ci' is the identifier of the class of which attribute 'ai' is a member and 'protection' is either "public", "protected" or "private" and 'name' is a valid C++ attribute name.

The CSI1 mathematical relational calculus notation measure definition is interpreted as:

> Measure CIS1 is the set of element pairs (ci, total) which is the set of element pairs (x.ci, total) where 'x' is an element of the set of classes C, and 'total' is the number of elements in the set of all y.ai where 'y' is an element of the set of attributes A, and y.ci equals x.ci and y.protection is either 'public' or 'protected'.

The example below shows the correspondence between relational calculus notation measure definition CIS1 and a database SQL query. The join between relations C and A is highlighted in bold in the measure definition and implemented in the WHERE statement of the SQL query.

CIS1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ai | y ∈ A ∧ **y.ci = x.ci** ∧ y.protection ∈ {public, protected}}}

```
SELECT C.ci, Count(SELECT A.ai FROM A GROUP BY A.ai HAVING
((A.protection)="public" Or (A.protection)="protected")) AS total
WHERE C.ci = A.ci
GROUP BY C.ci;
```

Using relational calculus notation (Grassmann and Tremblay 1996, pp. 620-624), and the mathematical entity models developed in section 4-2 of Chapter 4, descriptive measures of C++ class and object modularity are defined.

5.2.3 Measures of C++ class modularity

In Chapter 3, interface dependence, external relationships, connection obscurity and dependency are identified as important sub-characteristics of object-oriented software modularity. In Chapter 4, natural language and mathematical entity models are defined to describe ways in which these modularity sub-characteristics are manifest in C++ classes and objects. In this chapter, measures are defined to describe the degree to which C++ class and object modules posses these modularity sub-characteristics. Each measure can be directly traced to the sub-characteristic it describes.

These links are described in two ways. Firstly, each measure definition is named by an acronym to reflect the sub-characteristic it describes. Also, point number of the natural language feature each measure quantifies is included with its definition. For example, measure CSI1 defined in the previous example describes feature 1.2.1 of the C++ class interface dependence natural language model. This feature directly describes **C**lass **I**nterface **S**ize, hence the measure is named CSI. The numbers used to distinguish between the different CSI measures have no special meaning. The second way in which the links between measures and sub-characteristics are described is through the CHARMER diagram. This diagram traces each measure, through the feature it quantifies, to the sub-characteristic it describes. This is useful when selecting

measures for a particular application and analysing the interpreting the resulting measured data. Alternatively, by traversing the diagram from characteristic to measure, it is possible to see which sub-characteristics are described by measures and which are not. This is useful when performing a content validation of the measure set.

Some measures, although describing separate sub-characteristics of modularity, are quantifying the same features. This issue was discussed in section 4.2.3.6 of Chapter 4. To identify measures quantifying the same features, the measure names are extended. For example, measures CIS1 and CDE1 quantify the same feature. This is indicated by extending their names to CIS1:CCM1 and CDE1:CCM1 to indicate they belong to the group of **C**lass **C**ommon **M**easure **1**. Measures describing common features are identified this way in the tables where they are defined. Tables 5-14 and 5-32 list the sets of measures describing common features.

5.2.3.1 Measures of C++ class interface dependence

Figure 3-3 shows that the interface dependence sub-characteristic of object oriented software modularity is sub-characterised, at the lowest level, as interface element interdependence, interface size and data exposure. The natural language and mathematical entity models of C++ class interface dependence are defined in section 4.2.3.2. These models describe software features that increase the levels of C++ class interface element interdependence, interface size and data exposure present in the software.

The following tables define measures that quantify the software features identified in the natural language entity models of class interface dependence. These measures describe the levels of interface element interdependence, interface size and data exposure present in a C++ class. The first row of each table indicates the particular sub-characteristic described by the measures. The left-hand column of the table states the measure name and identifies the natural language feature it quantifies. The right hand column defines the measure in natural language and mathematical terms.

Class Interface Dependence : **Interface Element Interdependence**	
CIEI1 Feature 1.1.1	For each class non-constructor or non-destructor interface method, the number of same class interface attributes it directly reads. CIEI1 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor} ∧ total = #{y.ai | y ∈ A ∧ y.protection ∈ {public, protected} ∧ x.ci = y.ci ∧ (x.mi, y.ai) ∈ MCReadA}}
CIEI2 Feature 1.1.1	For each class non-constructor or non-destructor interface method, the number of same class interface attributes it directly writes. CIEI2 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor} ∧ total = #{y.ai | y ∈ A ∧ y.protection ∈ {public, protected} ∧ x.ci = y.ci ∧ (x.mi, y.ai) ∈ MCWriteA}}
CIEI3 Feature 1.1.1	For each class non-constructor or non-destructor interface method, the number of same class interface attributes it directly both reads and writes. CIEI3 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor} ∧ total = #{y.ai | y ∈ A ∧ y.protection ∈ {public, protected} ∧ x.ci = y.ci ∧ (x.mi, y.ai) ∈ MCWriteA ∧ (x.mi, y.ai) ∈ MCReadA}}
CIEI4 Feature 1.1.2	For each class non-constructor or non-destructor interface method, the number of same class interface attributes it indirectly reads. CIEI4 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor} ∧ total = #{y.ai | y ∈ A ∧ y.protection ∈ {public, protected} ∧ x.ci = y.ci ∧ (x.mi, y.ai) ∈ MICReadA}}
CIEI5 Feature 1.1.2	For each class non-constructor or non-destructor interface method, the number of same class interface attributes it indirectly writes. CIEI5 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor} ∧ total = #{y.ai | y ∈ A ∧ y.protection ∈ {public, protected} ∧ x.ci = y.ci ∧ (x.mi, y.ai) ∈ MICWriteA}}

Table 5-1 Measures of C++ Class Interface Element Interdependence

Class Interface Dependence : **Interface Element Interdependence (cont.)**	
CIEI6 Feature 1.1.2	For each class non-constructor or non-destructor interface method, the number of same class interface attributes it indirectly both reads and writes. CIEI6 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor} ∧ total = #{y.ai | y ∈ A ∧ y.protection ∈ {public, protected} ∧ x.ci = y.ci ∧ (x.mi, y.ai) ∈ MICWriteA ∧ (x.mi, y.ai) ∈ MICReadA}}
CIEI7 Feature 1.1.3	For each class non-constructor or non-destructor interface method, the number of same class interface methods it directly invokes. CIEI7 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor} ∧ total = #{y.ai | y ∈ M ∧ y.protection ∈ {public, protected} ∧ x.ci = y.ci ∧ (x.mi, y.mi) ∈ MCInvM}}
CIEI8 Feature 1.1.4	For each class non-constructor or non-destructor interface method, the number of same class interface methods it indirectly invokes. CIEI8 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor} ∧ total = #{y.ai | y ∈ M ∧ y.protection ∈ {public, protected} ∧ x.ci = y.ci ∧ (x.mi, y.mi) ∈ MICInvM}}

Table 5-2 Measures of C++ Class Interface Element Interdependence (cont.)

Class Interface Dependence : Interface Implementation Dependence : **Interface Size**	
CIS1:CCM1 Feature 1.2.1	For each class, the number of interface attributes. CIS1:CCM1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ai | y ∈ A ∧ y.ci = x.ci ∧ y.protection ∈ {public, protected}}}
CIS2 Feature 1.2.1	For each class, the number of hidden attributes. CIS2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ai | y ∈ A ∧ y.ci = x.ci ∧ y.protection = private}}
CIS3 Feature 1.2.2	For each class, the number of non-constructor or non-destructor interface methods. CIS3 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.mi | y ∈ M ∧ y.ci = x.ci ∧ y.protection ∈ {public, protected} ∧ y.purpose ∉ {constructor, destructor}}}
CIS4 Feature 1.2.2	For each class, the number of hidden methods. CIS4 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.mi | y ∈ M ∧ y.ci = x.ci ∧ (y.protection = private)}}
CIS5 Feature 1.2.3.1	For each class non-constructor or non-destructor interface method, the total number of lines of code within the method. CIS5 = {(ci, mi, total)} = {(x.ci, x.mi, y.lines) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor}}
CIS6 Feature 1.2.3.2	For each class non-constructor or non-destructor interface method, the total number of same class, other methods directly invoked. CIS6 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor} ∧ total = #{z.mi | z ∈ M ∧ z.ci = x.ci ∧ z.mi ≠ x.mi ∧ (x.mi, z.mi) ∈ MCInvM}}
CIS7:CCM2 Feature 1.2.3.3	For each class non-constructor or non-destructor interface method, the total number of same class attributes directly read. CIS7:CCM2 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor} ∧ total = #{z.ai | z ∈ A ∧ z.ci = x.ci ∧ (x.mi, z.ai) ∈ MCReadA}}
CIS8:CCM3 Feature 1.2.3.3	For each class non-constructor or non-destructor interface method, the total number of same class attributes directly written. CIS8:CCM3 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor} ∧ total = #{z.ai | z ∈ A ∧ z.ci = x.ci ∧ (x.mi, z.ai) ∈ MCWriteA}}

Table 5-3 Measures of C++ Class Interface Size

Class Interface Dependence : Interface Implementation Dependence : **Data Exposure**	
CDE1:CCM1 Feature 1.3.1	For each class, the number of interface attributes. CDE1:CCM1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ai | y ∈ A ∧ y.ci = x.ci ∧ y.protection ∈ {public, protected}}}
CDE2:CCM2 Feature 1.3.2	For each class non-constructor or non-destructor interface method, the total number of same class attributes directly read. CDE2:CCM2 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor} ∧ total = #{z.ai | z ∈ A ∧ z.ci = x.ci ∧ (x.mi, z.ai) ∈ MCReadA}}
CDE3:CCM3 Feature 1.3.2	For each class non-constructor or non-destructor interface method, the total number of same class attributes directly written. CDE3:CCM3 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ M ∧ x.protection ∈ {public, protected} ∧ x.purpose ∉ {constructor, destructor} ∧ total = #{z.ai | z ∈ A ∧ z.ci = x.ci ∧ (x.mi, z.ai) ∈ MCWriteA}}
CDE4 Feature 1.3.3	For each class attribute, its level of protection and the number of same class, interface, non-constructor or non-destructor methods directly writing to it. CDE4 = {(ci, ai, protection, total)} = {(x.ci, x.ai, x.protection, total) | x ∈ A ∧ total = #{y.mi | y ∈ M ∧ x.ci = y.ci ∧ y.protection ∈ {public, protected} ∧ y.purpose ∉ {constructor, destructor} ∧ (y.mi, x.ai) ∈ MCWriteA}}
CDE5 Feature 1.3.3	For each class attribute, its level of protection and the number of same class, interface, non-constructor or non-destructor methods directly reading from it. CDE5 = {(ci, ai, protection, total)} = {(x.ci, x.ai, x.protection, total) | x ∈ A ∧ total = #{y.mi | y ∈ M ∧ x.ci = y.ci ∧ y.protection ∈ {public, protected} ∧ y.purpose ∉ {constructor, destructor} ∧ (y.mi, x.ai) ∈ MCReadA}}

Table 5-4 Measures of C++ Class Data Exposure

The previous Tables 5-1 to 5-6 unambiguously define each measure of class interface dependence and, by specifying a natural language entity model feature number, identify the particular aspect of modularity described by each measure. The CHARMER diagram supplements these tables by presenting the relationships between characteristics, features and measures in a way that can be more readily understood. Figure 5-3 illustrates the C++ class interface dependence sub-characteristic to measure relationships. In this CHARMER diagram, the sub-characteristics are identified by name, the natural language model features by point number and the descriptive measures by their acronym name. This diagram shows that all the identified C++ class interface dependence sub-characteristics have associated features that are

identified in the natural language model. It shows that all these identified features are described by at least one measure. This means that all the identified C++ class interface dependence sub-characteristics are described by their associated measures.

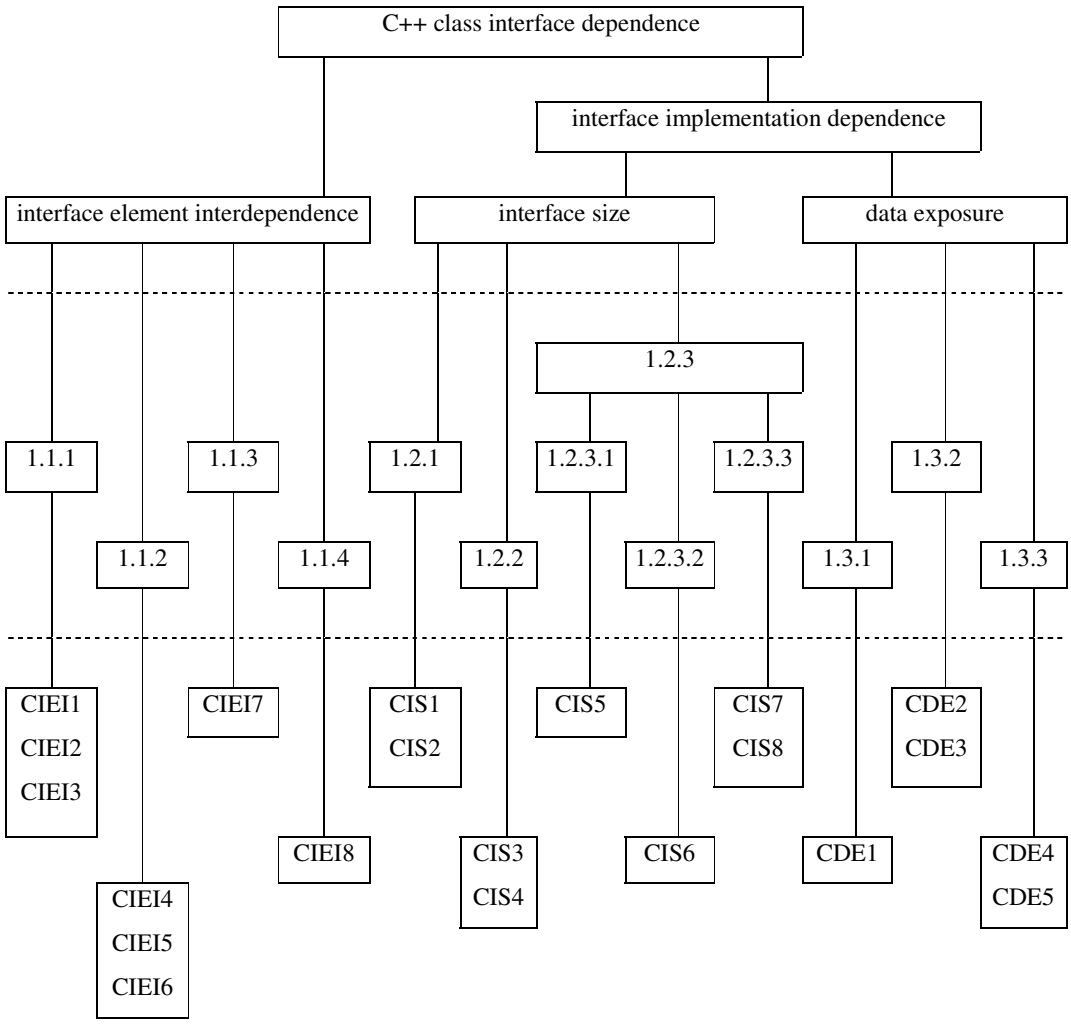


Figure 5-3 C++ class interface dependence characteristic to measure relationship (CHARMER) diagram

The following sections of this chapter present the operational definition of C++ class external relationships, connection obscurity and dependency modularity sub-characteristics, and C++ object interface dependence, external relationships, connection obscurity and dependency modularity sub-characteristics.

5.2.3.2 Measures of C++ class external relationships

Figure 3-3 shows that the external relationships sub-characteristic of object oriented software modularity is sub-characterised, at the lowest level, as external relationships within the measured software system and external relationships outside the measured software system. As previously discussed in section 3.2.3.3 of Chapter 3, measures will not be developed to describe external relationships outside the measured software system. The natural language and mathematical entity models of C++ class external relationships are defined in section 4.2.3.3. These models describe software features that increase the levels of C++ class external relationships present in the software.

The following table defines measures that quantify the software features identified in the natural language entity models of class external relationships.

Class External Relationships : **External Relationships**	
CER1 Feature 2.1.1	For each class, the number of immediate parent classes it has. This is the same for all hierarchies within which the class occurs. CER1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.parent_ci | y ∈ IP ∧ x.ci = y.ci}}
CER2 Feature 2.1.2	For each class, the number of distant ancestor classes it has. This is the same for all hierarchies within which the class occurs. CER2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ancestor_ci | y ∈ IDA ∧ x.ci = y.ci}}
CER3 Feature 2.1.3	For each class, the number of immediate child classes it has. CER3 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y. ci | y ∈ IP ∧ x.ci = y.parent_ci}}
CER4 Feature 2.1.4	For each class, the number of distant descendent classes it has. CER4 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ IDA ∧ x.ci = y.ancestor_ci}}
CER5:CCM4 Feature 2.1.5	For each class, the number of other classes that are immediate full or partial friends to the class. CER5:CCM4 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.friend_ci | y ∈ CEF ∧ x.ci = y.ci}}

Table 5-5 Measures of Class External Relationships

Class External Relationships : **External Relationships (cont.)**	
CER6:CCM5 Feature 2.1.6	For each class, the number of global functions that are immediate friends to the class. CER6:CCM5 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.friend_fi | y ∈ FF ∧ x.ci = y.ci}}
CER7:CCM6 Feature 2.1.7	For each class, the number of other classes to which it is an immediate full or partial friend. CER7:CCM6 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ CEF ∧ x.ci = y.friend_ci}}
CER8 Feature 2.1.8	For each class, the number of global functions within its scope. CER8 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.fi | y ∈ F ∧ ∃z{z | z ∈ SF ∧ z.ci = x.ci ∧ z.fi = y.fi}}}
CER9 Feature 2.1.9	For each class, the number of global variables within its scope. CER9 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.vi | y ∈ V ∧ ∃z{z | z ∈ SV ∧ z.ci = x.ci ∧ z.vi = y.vi}}}

Table 5-6 Measures of Class External Relationships (cont.)

Figure 5-4 illustrates the C++ class external relationships sub-characteristic to measure relationships. As previously discussed in section 3.2.3.3 of Chapter 3, measures will not be developed to describe external relationships outside the measured software system. This is reflected in the Figure 5-4 CHARMER diagram by the fact that no software features and measures are linked to this sub-characteristic. Features are identified for the remaining class external relationships sub-characteristic and Figure 5-4 shows that each of these software features is quantified by a class external relationships (CER) measure.

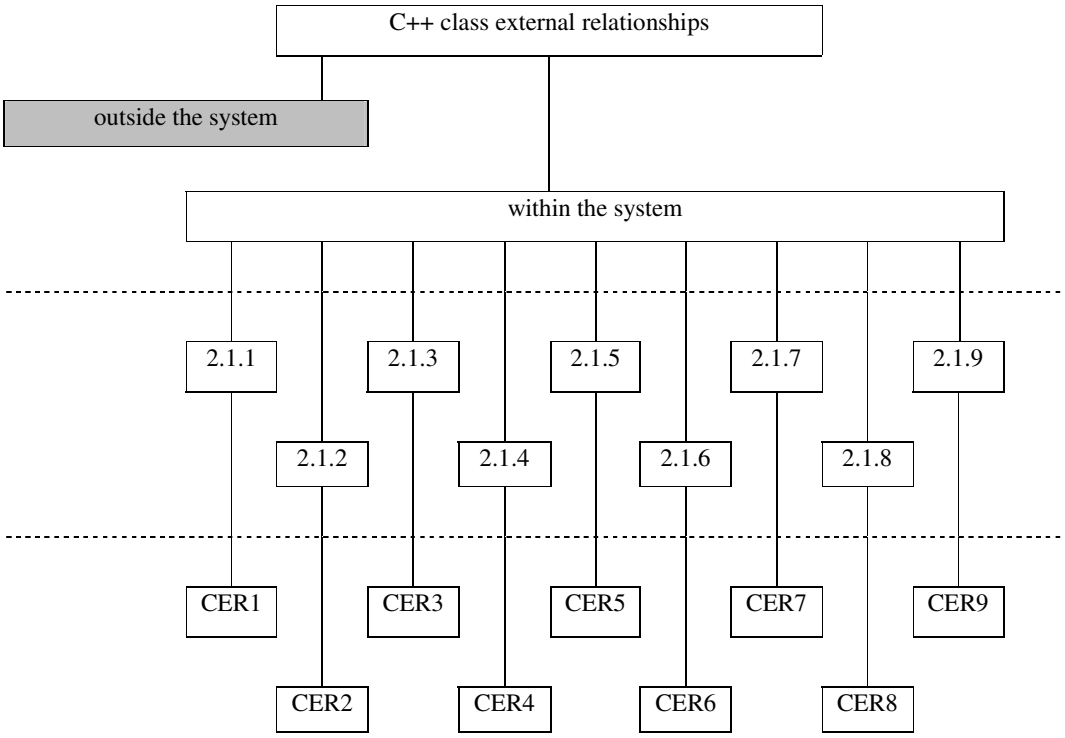


Figure 5-4 C++ class external relationships CHARMER diagram

5.2.3.3 Measures of C++ class connection obscurity

Figure 3-3 shows that the connection obscurity sub-characteristic of object oriented software modularity is sub-characterised at the lowest level as variable connection, unstated relationship, distant connection, unexpected relationship and connection via non-standard interface. The Chapter 4, section 4.2.3.4 natural language model of C++ class connection obscurity shows that the variable connection sub-characteristic of connection obscurity is not applicable to C++ class modularity.

The following tables define measures that quantify the software features identified in the natural language entity models of class connection obscurity.

Class Connection Obscurity : **Unstated Relationship**	
CUR1 Feature 3.2.1	For each class, the number of global variables its methods directly read from or write to. CUR1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.vi | y ∈ (MGReadV ∪ MGWriteV) ∧ ∃z{z | z ∈ M ∧ x.ci = z.ci ∧ z.mi = y.mi}}}
CUR2:CCM7 Feature 3.2.2	For each class, the number of global functions its methods directly invoke. CUR2:CCM7 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.fi | y ∈ MGInvF ∧ ∃z{z | z ∈ M ∧ x.ci = z.ci ∧ z.mi = y.mi}}}

Table 5-7 Measures of Class Unstated Relationship

Class Connection Obscurity : **Distant Connection**	
CDC1:CCM8 Feature 3.3.1	For each class, the number of other classes it is connected to via direct access to at least one shared global variable. CDC1:CCM8 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ M ∧ y.ci ≠ x.ci ∧ ∃z{z | z ∈ (MGReadV ∪ MGWriteV) ∧ z.mi = y.mi} ∧ ∃w{w | w ∈ M ∧ w.ci = x.ci ∧ (w.mi, z.vi) ∈ (MGReadV ∪ MGWriteV)}}
CDC2 Feature 3.3.2	For each class, the number of distant ancestor classes whose elements it directly accesses. CDC2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ancestor_ci | y ∈ IDA ∧ y.ci = x.ci ∧ ∃z{z ∈ M ∧ x.ci = z.ci} ∧ (∃m{m | m ∈ M ∧ m.ci = y.ancestor_ci} ∧ (z.mi, m.mi) ∈ MCInvM} ∨ ∃a{a| a ∈ A ∧ a.ci = y.ancestor_ci} ∧ (z.mi, a.ai) ∈ (MCReadA ∪ MCWriteA))}}

Table 5-8 Measures of Class Distant Connection

Class Connection Obscurity : Non-standard Connection : **Unexpected Relationship**	
CUER1:CCM8 Feature 3.4.1	For each class, the number of other classes it is connected to via direct access to at least one shared global variable. CUER1:CCM8 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ M ∧ y.ci ≠ x.ci ∧ ∃z{z | z ∈ (MGReadV ∪ MGWriteV) ∧ z.mi = y.mi} ∧ ∃w{w | w ∈ M ∧ w.ci = x.ci ∧ (w.mi, z.vi) ∈ (MGReadV ∪ MGWriteV)}}
CUER2:CCM4 Feature 3.4.2	For each class, the number of other classes that are immediate full or partial friends to the class. CUER2:CCM4 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.friend_ci | y ∈ CEF ∧ x.ci = y.ci}}
CUER3:CCM6 Feature 3.4.3	For each class, the number of other classes to which it is an immediate full or partial friend. CUER3:CCM6 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ CEF ∧ x.ci = y.friend_ci}}
CUER4:CCM5 Feature 3.4.4	For each class, the number of global functions that are immediate friends to the class. CUER4:CCM5 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.friend_fi | y ∈ FF ∧ x.ci = y.ci}}

Table 5-9 Measures of Class Unexpected Relationship

Class Connection Obscurity : Non-standard Connection : **Connection via Non-standard Interface**	
CNI1:CCM8 Feature 3.5.1	For each class, the number of other classes it is connected to via direct access to at least one shared global variable. CNI1:CCM8 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ M ∧ y.ci ≠ x.ci ∧ ∃z{z | z ∈ (MGReadV ∪ MGWriteV) ∧ z.mi = y.mi} ∧ ∃w{w | w ∈ M ∧ w.ci = x.ci ∧ (w.mi, z.vi) ∈ (MGReadV ∪ MGWriteV)}}
CNI2 Feature 3.5.2	For each class, the number of other classes it is connected to via direct access a member attribute of the other class. CNI2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ A ∧ y.ci ≠ x.ci ∧ ∃z{z | z ∈ (MCReadA ∪ MCWriteA) ∧ z.ai = y.ai} ∧ ∃w{w | w ∈ M ∧ w.ci = x.ci ∧ w.mi = z.mi}}}
CNI3 Feature 3.5.3	For each class, the number of other classes it is connected to via direct access of its own member attribute by the other class. CNI3 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ M ∧ y.ci ≠ x.ci ∧ ∃z{z | z ∈ (MCReadA ∪ MCWriteA) ∧ z.mi = y.mi} ∧ ∃w{w | w ∈ A ∧ w.ci = x.ci ∧ w.ai = z.ai}}}

Table 5-10 Measures of Class Connection via Non-standard Interface

Figure 5-5 illustrates the C++ class connection obscurity sub-characteristic to measure relationships. The Chapter 4, section 4.2.3.4 natural language model of C++ class connection obscurity shows that the variable connection sub-characteristic of connection obscurity is not applicable to C++ class modularity. This is reflected in the Figure 5-5 CHARMER diagram by the fact that no software features and measures are linked to this sub-characteristic. Features are identified for all the other class connection obscurity sub-characteristics and Figure 5-5 shows that each of these software features is quantified by a measure.

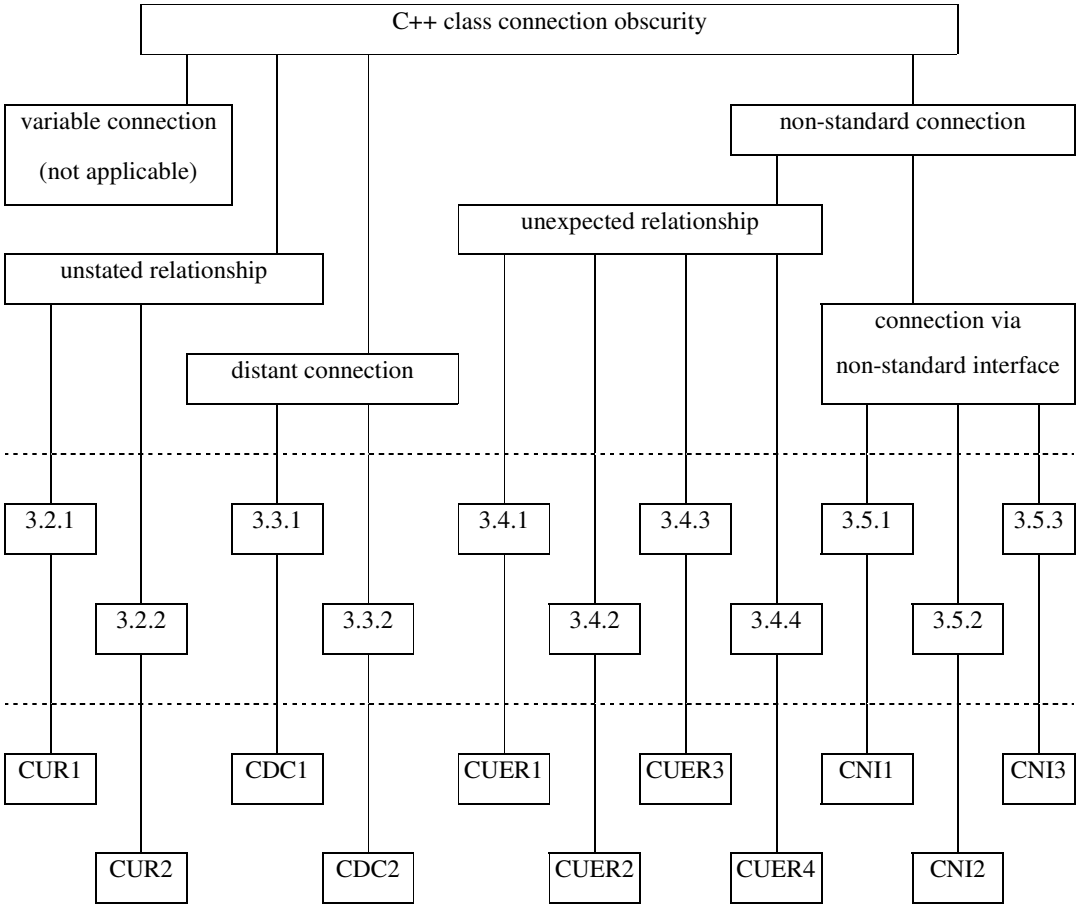


Figure 5-5 C++ class connection obscurity CHARMER diagram

5.2.3.4 Measures of C++ class dependency

Figure 3-3 shows that the dependency sub-characteristic of object oriented software modularity is sub-characterised at the lowest level as service invocation, interface provision, external variable reading and external function writing. The Chapter 4, section 4.2.3.5 natural language model of C++ class dependency shows that the interface provision sub-characteristic of dependency is not applicable to C++ class modularity.

The following tables define measures that quantify the software features identified in the natural language entity models of class dependency.

Class Dependency : **Service Invocation**	
CSI1:CCM7 Feature 4.1.1	For each class, the number of global functions its methods directly invoke. CSI1:CCM7 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.fi | y ∈ MGInvF ∧ ∃z{z | z ∈ M ∧ z.ci = x.ci ∧ z.mi = y.mi}}}
CSI2 Feature 4.1.2	For each class, the number of other classes whose methods are directly invoked. CSI2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ M ∧ y.ci ≠ x.ci ∧ ∃z{z | z ∈ M ∧ z.ci = x.ci ∧ (z.mi, y.mi) ∈ MCInvM}}}

Table 5-11 Measures of Class Service Invocation

Class Dependency : State Dependency : **External Variable Reading**	
CEVR1 Feature 4.3.1	For each class, the number of global variables its methods directly read from. CEVR1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.vi | y ∈ MGReadV ∧ ∃z{z | z ∈ M ∧ z.ci = x.ci ∧ z.mi = y.mi}}}
CEVR2 Feature 4.3.2	For each class, the number of other classes whose attributes its methods directly read from. CEVR2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ A ∧ x.ci ≠ y.ci ∧ ∃z{z | z ∈ M ∧ z.ci = x.ci} ∧ ∃w{w | w ∈ MCReadA ∧ w.mi = z.mi ∧ w.ai = y.ai}}}

Table 5-12 Measures of Class External Variable Reading

Class Dependency : State Dependency : **External Function Writing**	
CEFW1 Feature 4.4.1	For each class, the number of other classes writing to a global variable it directly reads from. CEFW1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ M ∧ x.ci ≠ y.ci ∧ ∃z{z | z ∈ MGWriteV ∧ y.mi = z.mi} ∧ ∃m{m | m ∈ M ∧ x.ci = m.ci} ∧ ∃w{w | w ∈ MGReadV ∧ w.mi = m.mi ∧ z.vi = w.vi}}}
CEFW2 Feature 4.4.2	For each class, the number of global functions writing to a global variable it directly reads from. CEFW2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.fi | y ∈ FGWriteV ∧ ∃z{z | z ∈ M ∧ x.ci = z.ci} ∧ ∃w{w | w ∈ MGReadV ∧ w.mi = z.mi ∧ y.vi = w.vi}}}
CEFW3 Feature 4.4.3	For each class, the number of other classes writing to a static attribute it directly reads from. CEFW3 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ M ∧ x.ci ≠ y.ci ∧ ∃z{z | z ∈ MCWriteA ∧ y.mi = z.mi} ∧ ∃m{m | m ∈ M ∧ x.ci = m.ci} ∧ ∃w{w | w ∈ MCReadA ∧ w.mi = m.mi ∧ z.ai = w.ai} ∧ ∃a{a | a ∈ A ∧ a.ai = z.ai ∧ a.static = true}}}
CEFW4 Feature 4.4.4	For each class, the number of other classes directly writing to one of its member attributes. CEFW4 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ M ∧ y.ci ≠ x.ci ∧ ∃z{z | z ∈ MCWriteA ∧ z.mi = y.mi} ∧ ∃w{w | w ∈ A ∧ w.ci = x.ci ∧ w.ai = z.ai}}}

Table 5-13 Measures of Class External Function Writing

Figure 5-6 illustrates the C++ class dependency sub-characteristic to measure relationships.

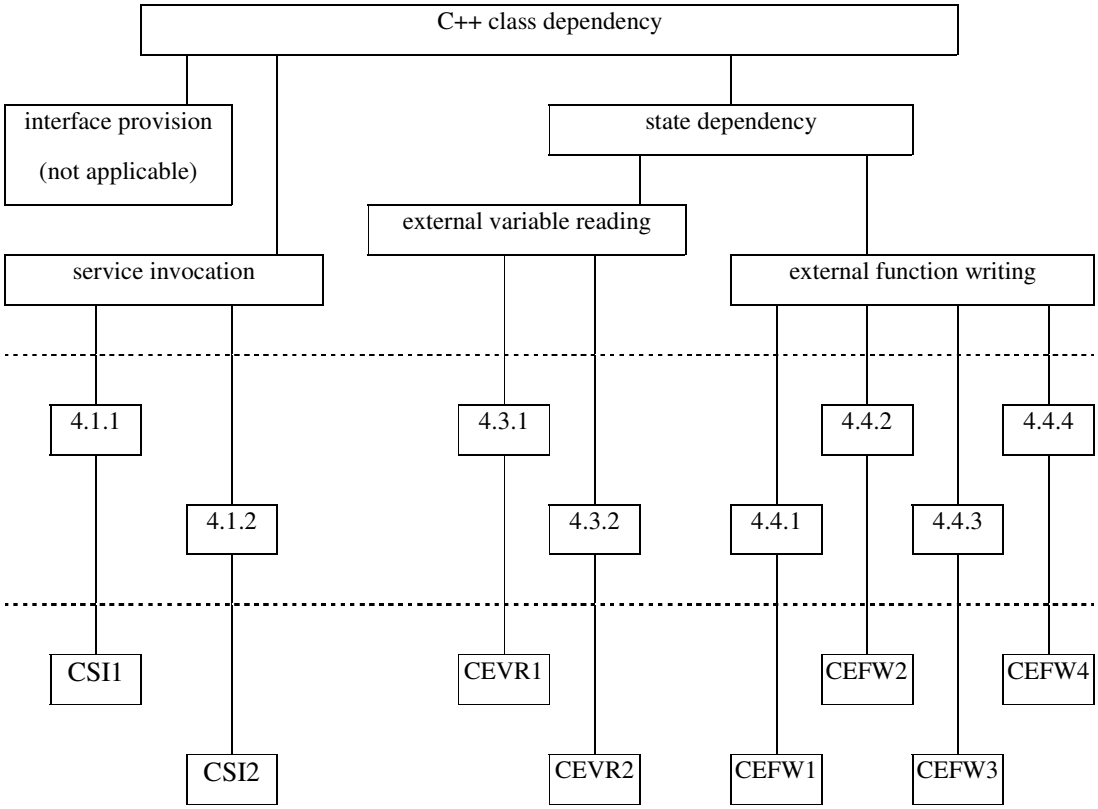


Figure 5-6 C++ class dependency CHARMER diagram

The Chapter 4, section 4.2.3.5 natural language model of C++ class dependency shows that the interface provision sub-characteristic of dependency is not applicable to C++ class modularity. This is reflected in the Figure 5-6 CHARMER diagram by the fact that no software features and measures are linked to this sub-characteristic. Features are identified for all the other class dependency sub-characteristics and Figure 5-6 shows that each of these software features is quantified by a measure.

Class Modularity Sub-characteristics	Common Feature	Measure
Interface Dependence : Interface Implementation Dependence: Interface Size	1.2.1	CIS1:CCM1
Interface Dependence : Interface Implementation Dependence: Data Exposure	1.3.1	CDE1:CCM1
Interface Dependence : Interface Implementation Dependence: Interface Size	1.2.3.3	CIS7:CCM2
Interface Dependence : Interface Implementation Dependence: Data Exposure	1.3.2	CDE2:CCM2
Interface Dependence : Interface Implementation Dependence: Interface Size	1.2.3.3	CIS8:CCM3
Interface Dependence : Interface Implementation Dependence: Data Exposure	1.3.2	CDE3:CCM3
External Relationships	2.1.5	CER5:CCM4
Connection Obscurity : Non-standard Connection : Unexpected Relationship	3.4.1	CUER2:CCM4
External Relationships	2.1.6	CER6:CCM5
Connection Obscurity : Non-standard Connection : Unexpected Relationship	3.4.4	CUER4:CCM5
External Relationships	2.1.7	CER7:CCM6
Connection Obscurity : Non-standard Connection : Unexpected Relationship	3.4.3	CUER3:CCM6
Connection Obscurity : Unstated Relationship	3.2.2	CUR2:CCM7
Dependency : Service Invocation	4.1.1	CSI1:CCM7
Connection Obscurity : Distant Connection	3.3.1	CDC1:CCM8
Connection Obscurity : Non-standard Connection : Unexpected relationship	3.4.1	CUER1:CCM8
Connection Obscurity : Non-standard Connection : Connection via Non-standard Interface	3.5.1	CNI1:CCM8

Table 5-14 Measures quantifying features common to several class modularity sub-characteristics

As discussed in Section 4.2.3.6 of Chapter 4, some features of the class modularity natural language model are common to several modularity sub-characteristics. Table 5-15 identifies the measures quantifying these common features. Including separate instances of common measured features in an analysis of measurement data provides a natural weighting of features having multiple effects on overall modularity. Where such multiple effects introduce unwanted dependencies between sub-characteristic descriptions, the information in Table 5-15 can be used to remove the measures that cause these dependencies.

5.2.4 Measures of C++ object modularity

The following section describe the operational definition of the C++ object modularity sub-characteristics based on the conceptual definitions of section 3.2.3 of Chapter 3 and the object entity models of section 4.2.4 of Chapter 4. For each major object modularity sub-characteristic, CHARMER diagrams describe the measures defined to quantify the software features that reduce the levels of modularity of a C++ object module. Some of the sub-characteristics, software features and measures in the object CHARMER diagrams are highlighted in grey, indicating an operational definition problem. Sub-characteristics and features that are highlighted are not described by any measures. Measures that are highlighted are unable, in some situations, to fully quantify their associated software features. Measure shortcomings are discussed individually for each of the object interface dependence, external relationships, connection obscurity and dependency sub-characteristics.

5.2.4.1 Measures of C++ object interface dependence

Figure 3-3 shows that the interface dependence sub-characteristic of object oriented software modularity is sub-characterised at the lowest level as service invocation, interface provision, external variable reading and external function writing. The natural language and mathematical entity models of C++ object interface dependence are defined in section 4.2.4.2. These models describe software features that increase the levels of C++ object interface dependence present in the software. The following tables define measures that quantify the software features identified in the natural language entity models of object interface dependence.

Object Interface Dependence : **Interface Element Interdependence**	
OIEI1 Feature 5.1.1	For each object-class non-constructor or non-destructor interface method, the number of same object-class interface attributes it directly reads. OIEI1 = {(ci, mi, total)} = {(x.ci, y.mi, total) | x ∈ OM ∧ y ∈ M ∧ x.mi = y.mi ∧ x.protection = public ∧ y.purpose ∉ {constructor, destructor} ∧ total = #{z.ai | z ∈ OA ∧ z.protection = public ∧ x.ci = z.ci ∧ (x.mi, z.ai) ∈ MCReadA}}
OIEI2 Feature 5.1.1	For each object-class non-constructor or non-destructor interface method, the number of same object-class interface attributes it directly writes. OIEI2 = {(ci, mi, total)} = {(x.ci, y.mi, total) | x ∈ OM ∧ y ∈ M ∧ x.mi = y.mi ∧ x.protection = public ∧ y.purpose ∉ {constructor, destructor} ∧ total = #{z.ai | z ∈ OA ∧ z.protection = public ∧ x.ci = z.ci ∧ (x.mi, z.ai) ∈ MCWriteA}}

Table 5-15 Measures of Object Interface Element Interdependence

Object Interface Dependence : **Interface Element Interdependence (cont.)**	
OIEI3 Feature 5.1.1	For each object-class non-constructor or non-destructor interface method, the number of same object-class interface attributes it directly both reads and writes. OIEI3 = {(ci, mi, total)} = {(x.ci, y.mi, total) | x ∈ OM ∧ y ∈ M ∧ x.mi = y.mi ∧ x.protection = public ∧ y.purpose ∉ {constructor, destructor} ∧ total = #{z.ai | z ∈ OA ∧ z.protection = public ∧ x.ci = z.ci ∧ (x.mi, z.ai) ∈ MCWriteA ∧ (x.mi, z.ai) ∈ MCReadA}}
OIEI4 Feature 5.1.2	For each object-class non-constructor or non-destructor interface method, the number of same object-class interface attributes it indirectly reads. OIEI4 = {(ci, mi, total)} = {(x.ci, y.mi, total) | x ∈ OM ∧ y ∈ M ∧ x.mi = y.mi ∧ x.protection = public ∧ y.purpose ∉ {constructor, destructor} ∧ total = #{z.ai | z ∈ OA ∧ z.protection = public ∧ x.ci = z.ci ∧ (x.mi, z.ai) ∈ MIOCReadA}}
OIEI5 Feature 5.1.2	For each object-class non-constructor or non-destructor interface method, the number of same object-class interface attributes it indirectly writes. OIEI5 = {(ci, mi, total)} = {(x.ci, y.mi, total) | x ∈ OM ∧ y ∈ M ∧ x.mi = y.mi ∧ x.protection = public ∧ y.purpose ∉ {constructor, destructor} ∧ total = #{z.ai | z ∈ OA ∧ z.protection = public ∧ x.ci = z.ci ∧ (x.mi, z.ai) ∈ MIOCWriteA}}
OIEI6 Feature 5.1.2	For each object-class non-constructor or non-destructor interface method, the number of same object-class interface attributes it indirectly both reads and writes. OIEI6 = {(ci, mi, total)} = {(x.ci, y.mi, total) | x ∈ OM ∧ y ∈ M ∧ x.mi = y.mi ∧ x.protection = public ∧ y.purpose ∉ {constructor, destructor} ∧ total = #{z.ai | z ∈ OA ∧ z.protection = public ∧ x.ci = z.ci ∧ (x.mi, z.ai) ∈ MIOCWriteA ∧ (x.mi, z.ai) ∈ MIOCReadA}}
OIEI7 Feature 5.1.3	For each object-class non-constructor or non-destructor interface method, the number of same object-class interface methods it directly invokes. OIEI7 = {(ci, mi, total)} = {(x.ci, y.mi, total) | x ∈ OM ∧ y ∈ M ∧ x.mi = y.mi ∧ x.protection = public ∧ y.purpose ∉ {constructor, destructor} ∧ total = #{z.ai | z ∈ OM ∧ w ∈ M ∧ z.mi = w.mi ∧ z.protection = public ∧ w.purpose ∉ {constructor, destructor} ∧ x.ci = z.ci ∧ (x.mi, z.mi) ∈ MCInvM}}
OIEI8 Feature 5.1.4	For each object-class non-constructor or non-destructor interface method, the number of same object-class interface methods it indirectly invokes. OIEI8 = {(ci, mi, total)} = {(x.ci, y.mi, total) | x ∈ OM ∧ y ∈ M ∧ x.mi = y.mi ∧ x.protection = public ∧ y.purpose ∉ {constructor, destructor} ∧ total = #{z.ai | z ∈ OM ∧ w ∈ M ∧ z.mi = w.mi ∧ z.protection = public ∧ w.purpose ∉ {constructor, destructor} ∧ x.ci = z.ci ∧ (x.mi, z.mi) ∈ MIOCInvM}}

Table 5-16 Measures of Object Interface Element Interdependence (cont.)

Object Interface Dependence : Interface Implementation Dependence : **Interface Size**	
OIS1:OCM1 Feature 5.2.1	For each object-class, the number of interface attributes. OIS1:OCM1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ai | y ∈ OA ∧ y.ci = x.ci ∧ y.protection = public}}
OIS2 Feature 5.2.1	For each object-class, the number of hidden attributes. OIS2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ai | y ∈ OA ∧ y.ci = x.ci ∧ y.protection ∈ (protected, private, inaccessible)}}
OIS3 Feature 5.2.2	For each object-class, the number of non-constructor or non-destructor interface methods. OIS3 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.mi | y ∈ OM ∧ y.ci = x.ci ∧ y.protection = public ∧ ∃z{z | z ∈ M ∧ z.mi = y.mi ∧ z.purpose ∉ {constructor, destructor}}}}
OIS4 Feature 5.2.2	For each object-class, the number of hidden methods. OIS4 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.mi | y ∈ OM ∧ y.ci = x.ci ∧ y.protection ∈ {protected, private, inaccessible}}}
OIS5 Feature 5.2.3.1	For each object-class non-constructor or non-destructor interface method, the total number of lines of code within the method. OIS5 = {(ci, mi, total)} = {(x.ci, x.mi, z.lines) | x ∈ OM ∧ z ∈ M ∧ x.protection = public ∧ x.mi = z.mi ∧ z.purpose ∉ {constructor, destructor}}
OIS6 Feature 5.2.3.2	For each object-class non-constructor or non-destructor interface method, the total number of same object-class, other methods directly invoked. OIS6 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ OM ∧ x.protection = public ∧ ∃z{z | z ∈ M ∧ z.mi = x.mi ∧ z.purpose ∉ {constructor, destructor}} ∧ total = #{m.mi | m ∈ OM ∧ m.ci = x.ci ∧ m.mi ≠ x.mi ∧ (x.mi, m.mi) ∈ MCInvM}}
OIS7:OCM2 Feature 5.2.3.3	For each object-class non-constructor or non-destructor interface method, the total number of same object-class attributes directly read. OIS7:OCM2 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ OM ∧ x.protection = public ∧ ∃z{z | z ∈ M ∧ z.mi = x.mi ∧ z.purpose ∉ {constructor, destructor}} ∧ total = #{a.ai | a ∈ OA ∧ a.ci = x.ci ∧ (x.mi, a.ai) ∈ MCReadA}}
OIS8:OCM3 Feature 5.2.3.3	For each object-class non-constructor or non-destructor interface method, the total number of same object-class attributes directly written. OIS8:OCM3 = {(ci, mi, total)} = {(x.ci, x.mi, total) | x ∈ OM ∧ x.protection = public ∧ ∃z{z | z ∈ M ∧ z.mi = x.mi ∧ z.purpose ∉ {constructor, destructor}} ∧ total = #{a.ai | a ∈ OA ∧ a.ci = x.ci ∧ (x.mi, a.ai) ∈ MCWriteA}}

Table 5-17 Measures of Object Interface Size

Object Interface Dependence : Interface Implementation Dependence : **Data Exposure**	
ODE1:OCM1 Feature 5.3.1	For each object-class, the number of interface attributes. ODE1:OCM1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ai | y ∈ OA ∧ y.ci = x.ci ∧ y.protection = public}}
ODE2:OCM2 Feature 5.3.2	For each object-class non-constructor or non-destructor interface method, the total number of same object-class attributes directly read. ODE2:OCM2 = {(ci, mi, total)} = {(x.ci, y.mi, total) | x ∈ OM ∧ x.protection = public ∧ y ∈ M ∧ x.mi = y.mi ∧ y.purpose ∉ {constructor, destructor}} ∧ total = #{a.ai | a ∈ OA ∧ a.ci = x.ci ∧ (x.mi, a.ai) ∈ MCReadA}}
ODE3:OCM3 Feature 5.3.2	For each object-class non-constructor or non-destructor interface method, the total number of same object-class attributes directly written. ODE3:OCM3 = {(ci, mi, total)} = {(x.ci, y.mi, total) | x ∈ OM ∧ x.protection = public ∧ y ∈ M ∧ x.mi = y.mi ∧ y.purpose ∉ {constructor, destructor}} ∧ total = #{a.ai | a ∈ OA ∧ a.ci = x.ci ∧ (x.mi, a.ai) ∈ MCWriteA}}
ODE4 Feature 5.3.3	For each object-class attribute, its level of protection and the number of same class, interface, non-constructor or non-destructor methods directly writing to it. ODE4 = {(ci, ai, protection, total)} = {(x.ci, x.ai, x.protection, total) | x ∈ OA ∧ total = #{y.mi | y ∈ OM ∧ x.ci = y.ci ∧ y.protection = public ∧ ∃z{z | z ∈ M ∧ z.mi = y.mi ∧ z.purpose ∉ {constructor, destructor}} ∧ (y.mi, x.ai) ∈ MCWriteA}}
ODE5 Feature 5.3.3	For each object-class attribute, its level of protection and the number of same class, interface, non-constructor or non-destructor methods directly reading from it. ODE5 = {(ci, ai, protection, total)} = {(x.ci, x.ai, x.protection, total) | x ∈ OA ∧ total = #{y.mi | y ∈ OM ∧ x.ci = y.ci ∧ y.protection = public ∧ ∃z{z | z ∈ M ∧ z.mi = y.mi ∧ z.purpose ∉ {constructor, destructor}} ∧ (y.mi, x.ai) ∈ MCReadA}}

Table 5-18 Measures of Object Data Exposure

Figure 5-7 illustrates the C++ object interface dependence sub-characteristic to measure relationships. As previously discussed in Chapter 4, section 4.2.4.6, the mathematical entity model of object interface dependence is unable to describe the member elements of all possible C++ objects. Figure 5-9 shows that all the interface dependence measures are affected by this modelling shortcoming and that measure CER2 can be used to identify objects that are potentially only partially described by these measures. Table 4-3 of Chapter 4 describes how to further examine these identified objects to determine whether or not the measures of object interface dependence are able to fully describe them.

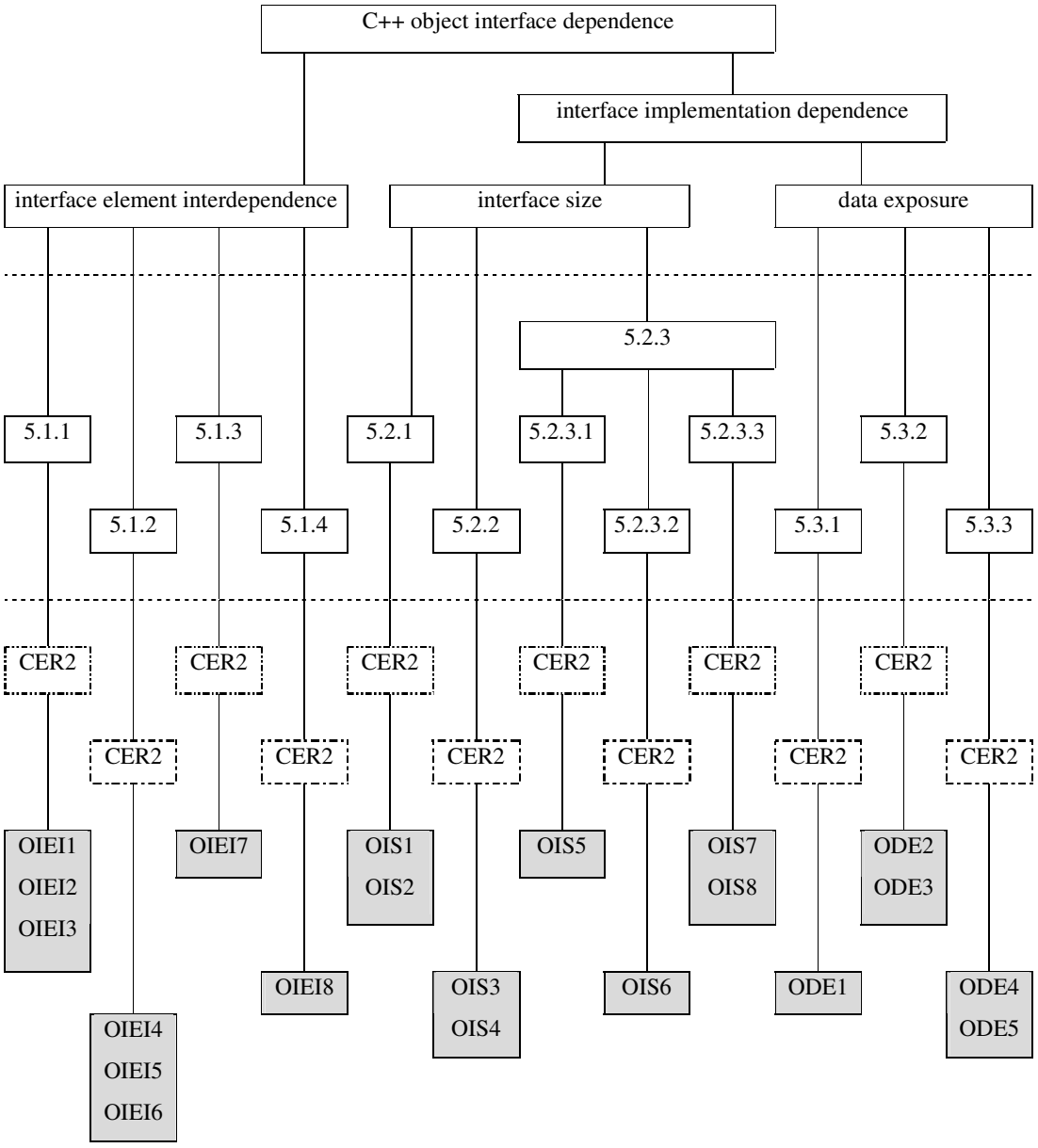


Figure 5-7 C++ object interface dependence CHARMER diagram

5.2.4.2 Measures of C++ object external relationships

Figure 3-3 shows that the external relationships sub-characteristic of object oriented software modularity is sub-characterised, at the lowest level, as external relationships within the measured software system and external relationships outside the measured software system. As previously discussed in section 3.2.3.3 of Chapter 3, measures will not be developed to describe external relationships outside the measured software system. The natural language and mathematical entity models of C++ object external relationships are defined in section 4.2.4.3. These models describe software features that increase the levels of C++ object external relationships present in the software.

The following table defines measures that quantify the software features identified in the natural language entity models of object external relationships.

Object External Relationships : **External Relationships**	
OER1 Feature 6.1.1	For each object-class, the number of immediate supplier object declarations. OER1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.oi | y ∈ IMO ∧ x.ci = y.ci}}
OER2 Feature 6.1.2	For each object-class, the number of global supplier objects within its scope. OER2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.oi | y ∈ O ∧ (y.global = true) ∧ ∃z{z | z ∈ SO ∧ z.ci = x.ci ∧ z.oi = y.oi}}}
OER3 Feature 6.1.3	For each object-class, the number of other classes to which it has an immediate supplier type association relationship. OER3 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ IMO ∧ ∃z{z | z ∈ O ∧ x.ci = z.ci ∧ y.oi = z.oi}}}
OER4 Feature 6.1.4	For each object-class, the number of global functions to which it has an immediate supplier type association relationship. OER4 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.fi | y ∈ FIMO ∧ ∃z{z | z ∈ O ∧ x.ci = z.ci ∧ y.oi = z.oi}}}

Table 5-19 Measures of Object External Relationships

Object External Relationships : **External Relationships (cont.)**	
OER5:OCM4 Feature 6.1.5	For each object-class, the number of static attributes it has. OER5:OCM4 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ai | y ∈ OA ∧ y.ci = x.ci ∧ ∃z{z | z ∈ A ∧ z.ai = y.ai ∧ z.static = true}}}
OER6:OCM5 Feature 6.1.6	For each object-class, the number of other object-classes that are full or partial friends. OER6:OCM5 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.friend_ci | y ∈ (CEF ∪ CIF) ∧ x.ci = y.ci}}
OER7:OCM6 Feature 6.1.7	For each object-class, the number of global functions that are full or partial friends. OER7:OCM6 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.friend_fi | y ∈ (FF ∪ FIF) ∧ x.ci = y.ci}}
OER8:OCM7 Feature 6.1.8	For each object-class, the number of other object-classes to which it is a full or partial friend. OER8:OCM7 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ (CEF ∪ CIF) ∧ x.ci = y.friend_ci}}

Table 5-20 Measures of Object External Relationships (cont.)

Figure 5-8 illustrates the C++ object external relationships sub-characteristic to measure relationships. As previously discussed in section 3.2.3.2 of Chapter 3, measures will not be developed to describe external relationships outside the measured software system. This is reflected in the Figure 5.8 CHARMER diagram by the fact that no software features and measures are linked to this sub-characteristic. As previously discussed in Chapter 4, section 4.2.4.6, the mathematical entity model of object external relationships is unable to describe the member elements of all possible C++ objects. Figure 5-8 shows that external relationships measures OER1 and OER5:OCM4 are affected by this modelling shortcoming and that measure CER2 can be used to identify objects that are potentially only partially described by these measures. Table 4-3 of Chapter 4 describes how to further examine these identified objects to determine whether or not the OER1 and OER5:OCM4 measures of object external relationships are able to fully describe them.

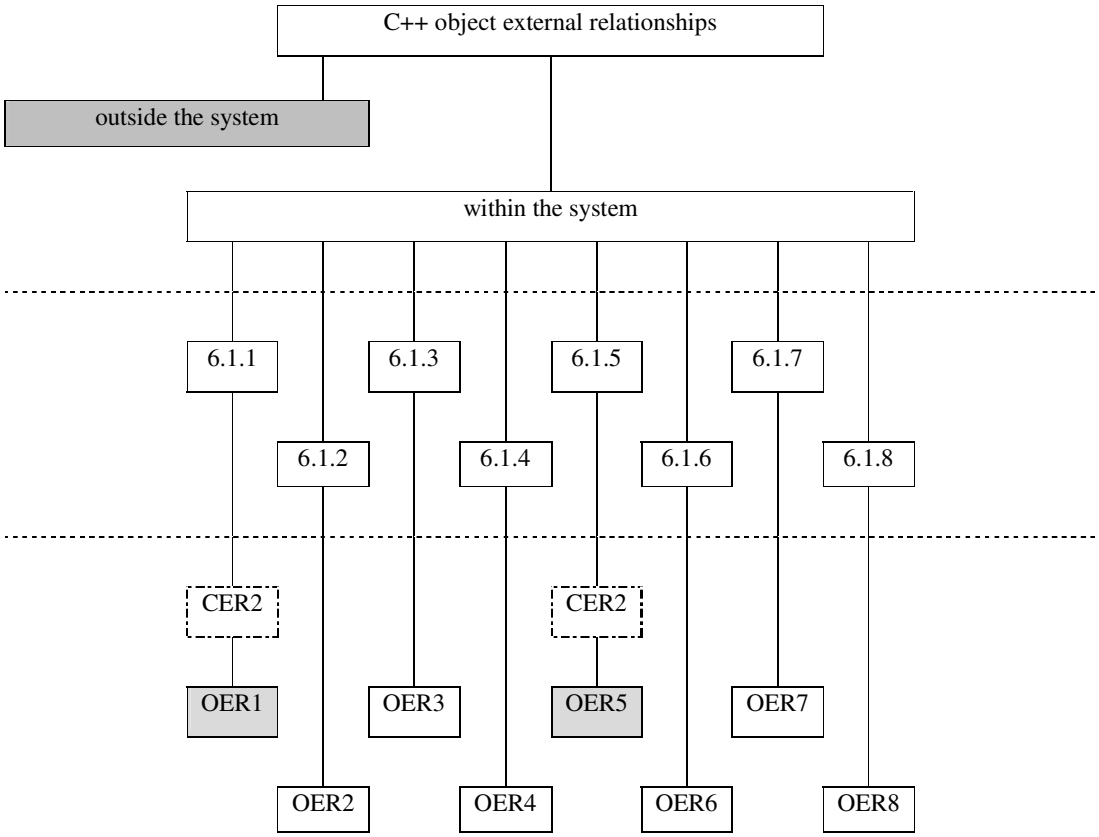


Figure 5-8 C++ object external relationships CHARMER diagram

5.2.4.3 Measures of C++ object connection obscurity

Figure 3-3 shows that the connection obscurity sub-characteristic of object oriented software modularity is sub-characterised at the lowest level as variable connection, unstated relationship, distant connection, unexpected relationship and connection via non-standard interface. The natural language and mathematical entity models of C++ object connection obscurity are defined in section 4.2.4.4. These models describe software features that increase the levels of C++ object connection obscurity present in the software.

The following tables define measures that quantify the software features identified in the natural language entity models of object connection obscurity.

Object Connection Obscurity : **Variable Connection**	
OVC1 Feature 7.1.1	For each object-class, the number of supplier objects declared as a pointer. OVC1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.oi | y ∈ IMO ∧ ∃z{z | z ∈ O ∧ z.oi = y.oi ∧ z.pointer = true}}}
OVC2 Feature 7.1.2	For each object-class, the number of pointer type global supplier objects within its scope. OVC2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.oi | y ∈ O ∧ y.global = true ∧ y.pointer = true ∧ ∃z{z ∈ SO ∧ z.ci = x.ci ∧ z.oi = y.oi}}}

Table 5-21 Measures of Object Variable Connection

Object Connection Obscurity : **Unstated Relationship**	
OUR1:OCM8 Feature 7.2.1	For each object-class, the number of global variables its methods directly read from or write to. OUR1:OCM8 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.vi | y ∈ (MGReadV ∪ MGWriteV) ∧ ∃z{z | z ∈ OM ∧ z.ci = x.ci ∧ z.mi = y.mi}}}
OUR2:OCM9 Feature 7.2.2	For each object-class, the number of global functions its methods directly invoke. OUR2:OCM9 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.fi | y ∈ MGInvF ∧ ∃z{z | z ∈ OM ∧ z.ci = x.ci ∧ z.mi = y.mi}}}
OUR3 Feature 7.2.3	For each object-class, the number of global objects its methods directly access. OUR3 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.oi | y ∈ MOAccessO ∧ ∃z{z | z ∈ OM ∧ z.ci = x.ci ∧ z.mi = y.mi} ∧ ∃w{w | w ∈ O ∧ w.oi = y.oi ∧ w.global = true}}}
OUR4 Feature 7.2.4	For each object-class, the number of inherited static attributes it has. OUR4 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ai | y ∈ (AA ∪ IAA) ∧ y.ci = x.ci ∧ ∃z{z | z ∈ A ∧ z.ai = y.ai ∧ z.static = true}}}
OUR5 Feature 7.2.5	For each object-class, the number of immediate parent or distant ancestor classes with immediate friend classes. OUR5 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #({y.parent_ci | y ∈ IP ∧ y.ci = x.ci ∧ ∃z{z | z ∈ CEF ∧ z.ci = y.parent_ci}} ∪ {y.ancestor_ci | y ∈ IDA ∧ y.ci = x.ci ∧ ∃z{z | z ∈ CEF ∧ z.ci = y.ancestor_ci}})}
OUR6 Feature 7.2.6	For each object-class, the number of immediate parent or distant ancestor classes with immediate friend global functions. OUR6 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #({y.parent_ci | y ∈ IP ∧ y.ci = x.ci ∧ ∃z{z | z ∈ FF ∧ z.ci = y.parent_ci}} ∪ {y.ancestor_ci | y ∈ IDA ∧ y.ci = x.ci ∧ ∃z{z | z ∈ FF ∧ z.ci = y.ancestor_ci}})}

Table 5-22 Measures of Object Unstated Relationship

Object Connection Obscurity : **Distant Connection**	
ODC1:OCM10 Feature 7.3.1	For each object-class, the number of other object-classes it is connected to via direct access to at least one shared global variable. ODC1:OCM10 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ OM ∧ x.ci ≠ y.ci ∧ ∃z{z | z ∈ OM ∧ z.ci = x.ci} ∧ ∃v{v | v ∈ V ∧ (y.mi, v.vi) ∈ (MGReadV ∪ MGWriteV) ∧ (z.mi, v.vi) ∈ (MGReadV ∪ MGWriteV)}}}
ODC2 Feature 7.3.2	For each object-class, the number of non-global objects it directly accesses where it does not have a direct member or inherited association relationship with the objects. Unable to define ODC2 in terms of previously defined mathematical model of object connection obscurity as mathematical model does not support the relationship “object member methods directly access non-global, non-directly associated object methods or attributes”.

Table 5-23 Measures of Object Distant Connection

Object Connection Obscurity : Non-standard Connection : **Unexpected Relationship**	
OUER1:OCM8 Feature 7.4.1	For each object-class, the number of global variables its methods directly access. OUER1:OCM8 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.vi | y ∈ (MGReadV ∪ MGWriteV) ∧ ∃z{z | z ∈ OM ∧ z.ci = x.ci ∧ z.mi = y.mi}}}
OUER2:OCM10 Feature 7.4.2	For each object-class, the number of other object-classes it is connected to via direct access to at least one shared global variable. OUER2:OCM10 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ OM ∧ x.ci ≠ y.ci ∧ ∃z{z | z ∈ OM ∧ z.ci = x.ci} ∧ ∃v{v | v ∈ V ∧ (y.mi, v.vi) ∈ (MGReadV ∪ MGWriteV) ∧ (z.mi, v.vi) ∈ (MGReadV ∪ MGWriteV)}}}
OUER3:OCM5 Feature 7.4.3	For each object-class, the number of other object-classes that are full or partial friends. OUER3:OCM5 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.friend_ci | y ∈ (CEF ∪ CIF) ∧ x.ci = y.ci}}
OUER4:OCM6 Feature 7.4.4	For each object-class, the number of global functions that are full or partial friends. OUER4:OCM6 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.friend_fi | y ∈ (FF ∪ FIF) ∧ x.ci = y.ci}}
OUER5:OCM7 Feature 7.4.5	For each object-class, the number of other object-classes to which it is a full or partial friend. OUER5:OCM7 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ (CEF ∪ CIF) ∧ x.ci = y.friend_ci}}

Table 5-24 Measures of Object Unexpected Relationship

Object Connection Obscurity : Non-standard Connection : **Unexpected Relationship (cont.)**	
OUER6:OCM4 Feature 7.4.6	For each object-class, the number of static attributes it has. OUER6:OCM4 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ai | y ∈ OA ∧ y.ci = x.ci ∧ ∃z{z | z ∈ A ∧ z.ai = y.ai ∧ z.static = true}}}
OUER7 Feature 7.4.6	For each object-class, the number of other object-classes it shares one or more static attributes with. OUER7 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ OA ∧ x.ci ≠ y.ci ∧ ∃z{z | z ∈ A ∧ y.ai = z.ai ∧ z.static = true} ∧ ∃a{a | a ∈ OA ∧ a.ci = x.ci ∧ a.ai = y.ai}}}

Table 5-25 Measures of Object Unexpected Relationship (cont.)

Object Connection Obscurity : Non-standard Connection : **Connection via Non-standard Interface**	
ONI1:OCM10 Feature 7.5.1	For each object-class, the number of other object-classes it is connected to via direct access to at least one shared global variable. ONI1:OCM10 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ OM ∧ x.ci ≠ y.ci ∧ ∃z{z | z ∈ OM ∧ z.ci = x.ci} ∧ ∃v{v | v ∈ V ∧ (y.mi, v.vi) ∈ (MGReadV ∪ MGWriteV) ∧ (z.mi, v.vi) ∈ (MGReadV ∪ MGWriteV)}}}
ONI2 Feature 7.5.2	For each object-class, the number of objects it is connected to via direct access to at least one of the object's attributes. ONI2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.oi | y ∈ MOAccessO ∧ y.action ∈ {read, write} ∧ ∃z{z | z ∈ OM ∧ z.ci = x.ci ∧ z.mi = y.mi}}}
ONI3 Feature 7.5.5	For each object-class, the number of objects instantiated from the object-class, whose attributes are directly accessed by another object-class. ONI3 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.oi | y ∈ MOAccessO ∧ y.action ∈ {read, write} ∧ ∃z{z | z ∈ O ∧ z.oi = y.oi ∧ z.ci = x.ci}}}
ONI4 Feature 7.5.6	For each object-class, the number of objects instantiated from the object-class whose attributes are directly accessed by a global function. ONI4 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.oi | y ∈ FOAccessO ∧ y.action ∈ {read, write} ∧ ∃z{z | z ∈ O ∧ z.oi = y.oi ∧ z.ci = x.ci}}}
ONI5 Feature 7.5.3	For each object-class, the number of other object classes it is connected to via direct access to at least one of the object's hidden attributes or methods. Unable to define ONI5 in terms of previously defined mathematical model of object connection obscurity as mathematical model does not support the relationship "object member methods directly access hidden object methods or attributes".

Table 5-26 Measures of Object Connection via Non-standard Interface

Object Connection Obscurity : Non-standard Connection : **Connection via Non-standard Interface (cont.)**	
ONI6 Feature 7.5.4	For each object-class, the number of other object classes it is connected to via direct access to at least one of the object's inaccessible attributes or methods. Unable to define ONI6 in terms of previously defined mathematical model of object connection obscurity as mathematical model does not support the relationship "object member methods directly access inaccessible object methods or attributes".
ONI7 Feature 7.5.7	For each object-class, the number of objects instantiated from the object-class whose hidden methods or attributes are directly accessed by the member methods of another object-class. Unable to define ONI7 in terms of previously defined mathematical model of object connection obscurity as mathematical model does not support the relationship "object member methods directly access hidden object methods or attributes".
ONI8 Feature 7.5.8	For each object-class, the number of objects instantiated from the object-class whose inaccessible methods or attributes are directly accessed by the member methods of another object-class. Unable to define ONI8 in terms of previously defined mathematical model of object connection obscurity as mathematical model does not support the relationship "object member methods directly access inaccessible object methods or attributes".
ONI9 Feature 7.5.9	For each object-class, the number of objects instantiated from the object-class whose hidden methods or attributes are directly accessed by a global function. Unable to define ONI9 in terms of previously defined mathematical model of object connection obscurity as mathematical model does not support the relationship "global function directly accesses hidden object methods or attributes".
ONI10 Feature 7.5.10	For each object-class, the number of objects instantiated from the object-class whose inaccessible methods or attributes are directly accessed by a global function. Unable to define ONI10 in terms of previously defined mathematical model of object connection obscurity as mathematical model does not support the relationship "global function directly accesses inaccessible object methods or attributes".

Table 5-27 Measures of Object Connection via Non-standard Interface (cont.)

Figures 5-9 and 5-10 link the natural language model of C++ object connection obscurity to the descriptive measures. As previously discussed in Chapter 4, section 4.2.4.6, the mathematical entity model of object connection obscurity is unable to describe the member elements of all possible C++ objects. Figures 5-9 and 5-10 show that connection obscurity measures OVC1,

OUR4, OUR5, OUR6 and OUER6:OCM4 are affected by this modelling shortcoming Measure CER2 can be used to identify objects that are potentially only partially described by these measures. The mathematical model of object connection obscurity does not describe point 7.3.2 of the object connection obscurity natural language model. Table 4-3 of Chapter 4 describes how to further examine these identified objects to determine whether or not the OVC1, OUR4, OUR5, OUR6 and OUER6:OCM4 measures of object connection obscurity are able to fully describe them. Section 4.2.4.6 of Chapter 4 also discusses the reasons that connection via non-standard interface features 7.5.3, 7.5.4, 7.5.7, 7.5.8, 7.5.9 and 7.5.10 are not described by measures. Figure 5-10 shows that measures OER6:OCM5, OER7:OCM6 and OER8:OCM7 can be used to identify objects whose measured description of connection obscurity is potentially affected by this loss of measure detail. Table 4-3 of Chapter 4 describes how to further examine these identified objects to determine whether or not they posses the natural language features 7.5.3, 7.5.4, 7.5.7, 7.5.8, 7.5.9 and 7.5.10 that are not described by measures.

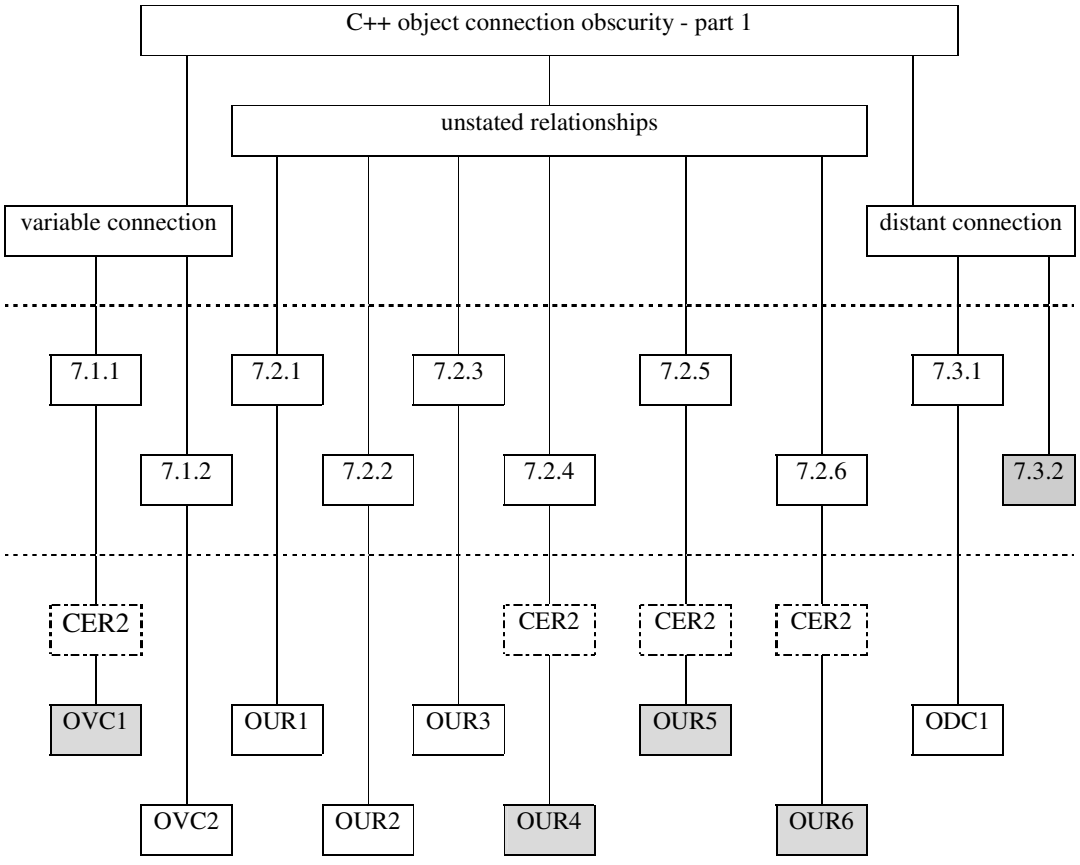


Figure 5-9 C++ object connection obscurity CHARMER diagram - Part 1

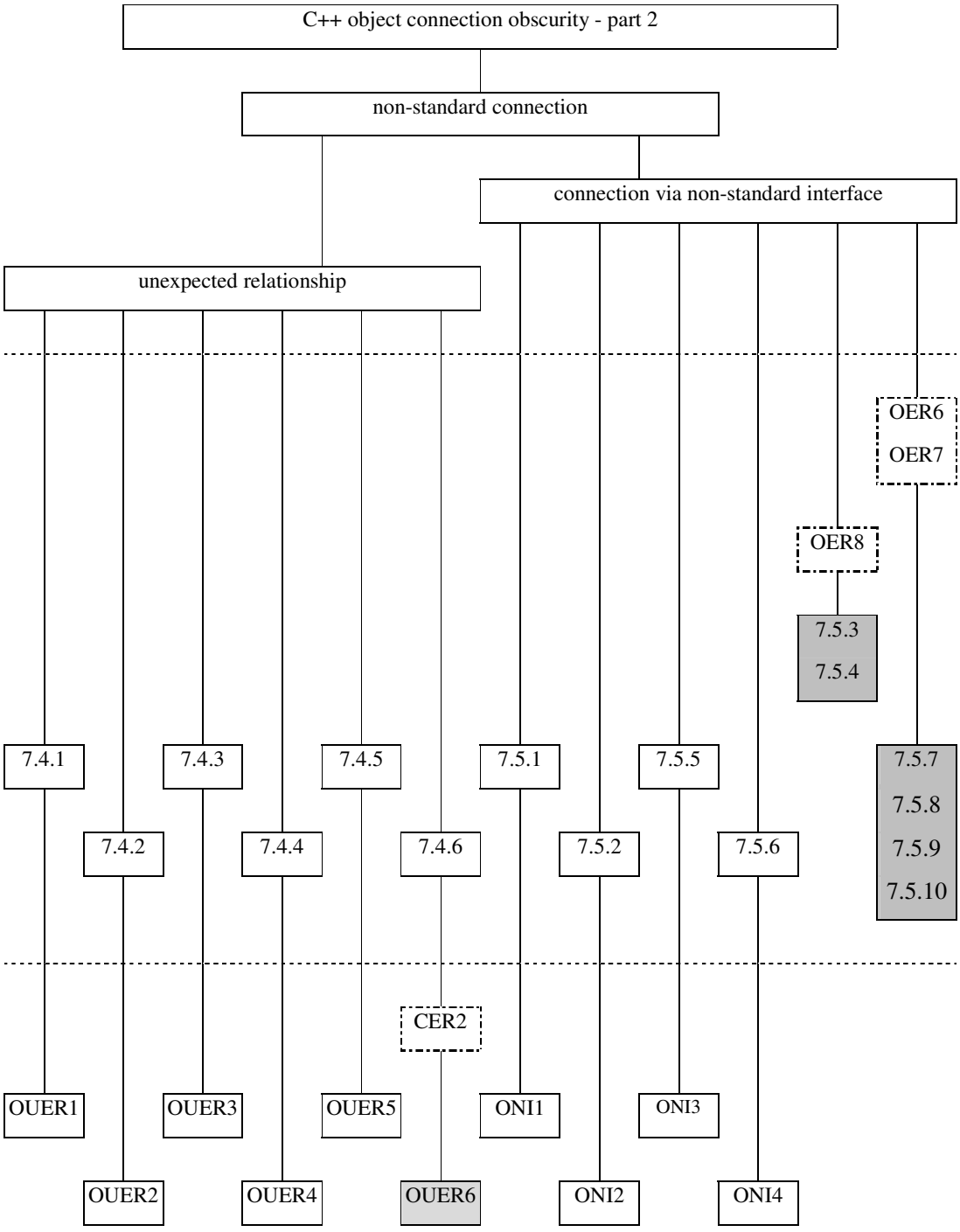


Figure 5-10 C++ object connection obscurity CHARMER diagram - Part 2

5.2.4.4 Measures of C++ object dependency

Figure 3-3 shows that the dependency sub-characteristic of object oriented software modularity is sub-characterised at the lowest level as service invocation, interface provision, external variable reading and external function writing. The natural language and mathematical entity models of C++ object dependency are defined in section 4.2.4.5. These models describe software features that increase the levels of C++ object dependency present in the software.

The following tables define measures that quantify the software features identified in the natural language entity models of object dependency.

Object Dependency : **Service Invocation**	
OSI1:OCM9 Feature 8.1.1	For each object-class, the number of global functions directly invoked. OSI1:OCM9 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.fi | y ∈ MGInvF ∧ ∃z{z | z ∈ OM ∧ x.ci = z.ci ∧ z.mi = y.mi}}}
OSI2 Feature 8.1.2	For each object-class, the number of other objects whose methods are directly invoked. OSI2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.oi | y ∈ MOAccessO ∧ y.action = invoke ∧ ∃z{z | z ∈ OM ∧ x.ci = z.ci ∧ z.mi = y.mi}}}

Table 5-28 Measures of Object Service Invocation

Object Dependency : **Interface Provision**	
OIP1 Feature 8.2.1	For each object-class, the number of inherited interface methods. OIP1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.mi | y ∈ AM ∧ y.ci = x.ci ∧ y.protection = public}}
OIP2 Feature 8.2.1	For each object-class, the number of inherited interface attributes. OIP2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ai | y ∈ AA ∧ y.ci = x.ci ∧ y.protection = public}}

Table 5-29 Measures of Object Interface Provision

Object Dependency : State Dependency : **External Variable Reading**	
OEVR1 Feature 8.3.1	For each object-class, the number of global variables its methods directly read. OEVR1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.vi | y ∈ MGReadV ∧ ∃z{z | z ∈ OM ∧ z.ci = x.ci ∧ z.mi = y.mi}}}
OEVR2 Feature 8.3.2	For each object-class, the number of other objects whose attributes are directly read. OEVR2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.oi | y ∈ MOAccessO ∧ y.action = read ∧ ∃z{z | z ∈ OM ∧ x.ci = z.ci ∧ z.mi = y.mi}}}

Table 5-30 Measures of Object External Variable Reading

Object Dependency : State Dependency : **External Function Writing**	
OEFW1 Feature 8.4.1	For each object-class, the number of other object-classes writing to a global variable it reads. OEFW1 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ C ∧ y.ci ≠ x.ci ∧ ∃z{z | z ∈ OM ∧ z.ci = y.ci} ∧ ∃w{w | w ∈ OM ∧ w.ci = x.ci} ∧ ∃v{v | v ∈ V ∧ (w.mi, v.vi) ∈ MGReadV ∧ (z.mi, v.vi) ∈ MGWriteV}}}
OEFW2 Feature 8.4.2	For each object-class, the number of global functions writing to a global variable read by the object-class. OEFW2 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.vi | y ∈ FGWriteV ∧ ∃z{z | z ∈ OM ∧ z.ci = x.ci} ∧ ∃w{w | w ∈ MGReadV ∧ w.mi = z.mi ∧ w.vi = y.vi}}}
OEFW3 Feature 8.4.3	For each object-class, the number of objects instantiated from that class that have a global functions writing to one of their attributes. OEFW3 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.oi | y ∈ O ∧ y.ci = x.ci ∧ ∃z{z | z ∈ FOAccessO ∧ z.oi = y.oi ∧ z.action = write}}}
OEFW4 Feature 8.4.4	For each object-class, the number of other object-classes class-writing to a static attribute class-read by the object-class. OEFW4 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.ci | y ∈ C ∧ y.ci ≠ x.ci ∧ ∃z{z | z ∈ OM ∧ z.ci = x.ci} ∧ ∃w{w | w ∈ OM ∧ w.ci = y.ci} ∧ ∃a{a | a ∈ A ∧ a.static = true ∧ (z.mi, a.ai) ∈ MCReadA ∧ (w.mi, a,ai) ∈ MCWriteA}}}
OEFW5 Feature 8.4.6	For each object-class, the number of objects instantiated from that class that have another object-class object-writing to one of their attributes. OEFW5 = {(ci, total)} = {(x.ci, total) | x ∈ C ∧ total = #{y.oi | y ∈ O ∧ y.ci = x.ci ∧ ∃z{z | z ∈ MOAccessO ∧ z.oi = y.oi ∧ z.action = write}}}
OEFW6 Feature 8.4.5	For each object-class, the number of objects instantiated from another object-class whose member methods directly object-write to a static attribute directly object-read by the object-class. Unable to define OEFW6 in terms of previously defined mathematical model of object dependency as mathematical model does not support the relationships “object member methods directly object-read and object-write to object member attributes”.

Table 5-31 Measures of Object External Function Writing

Figure 5-11 links the natural language model of C++ object dependency to the descriptive measures. As previously discussed in Chapter 4, section 4.2.4.6, the mathematical entity model of object dependency is unable to describe the member elements of all possible C++ objects.

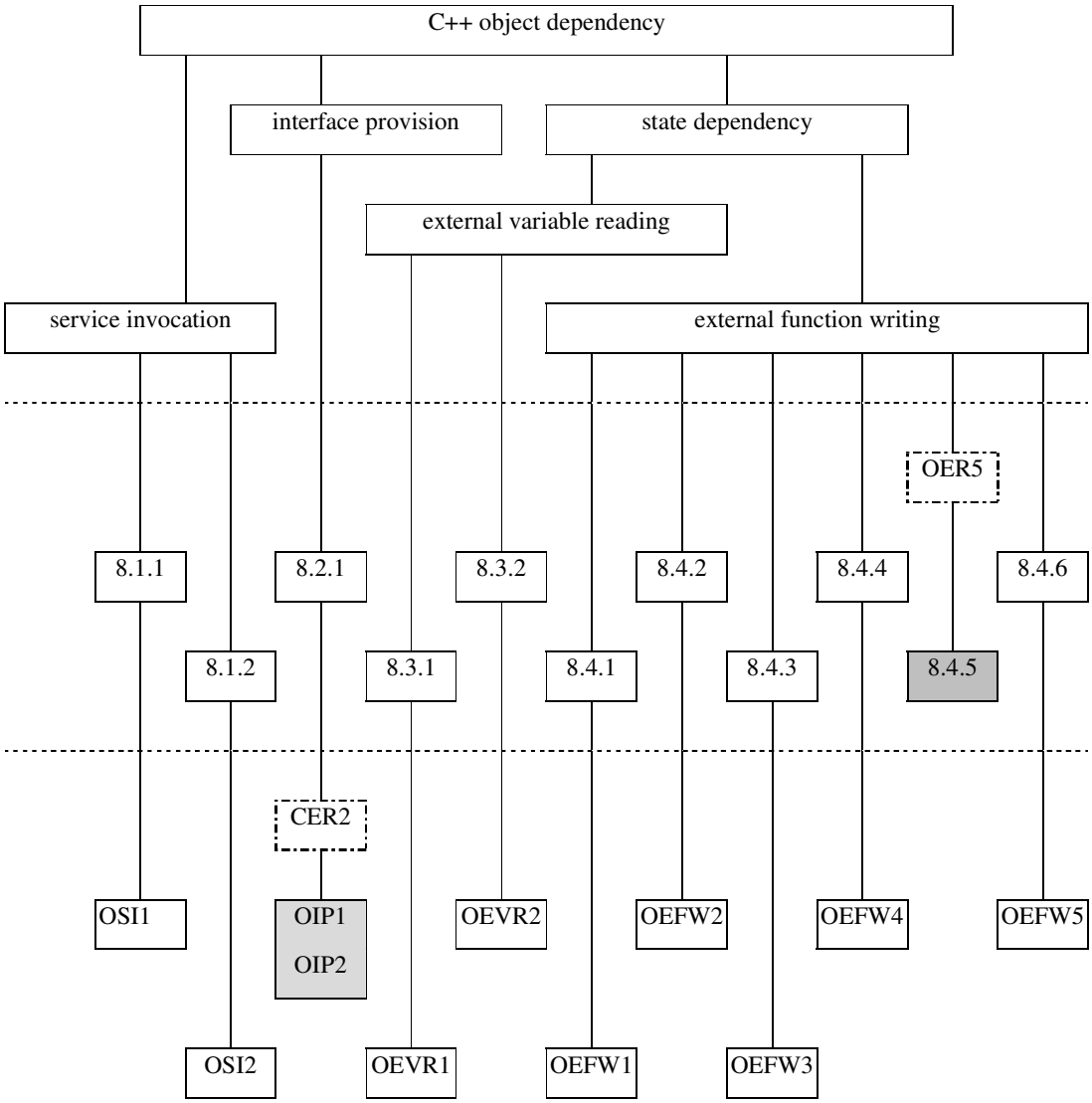


Figure 5-11 C++ object dependency CHARMER diagram

Figure 5-11 shows that dependency measures OIP1 and OIP2 are affected by this modelling shortcoming and measure CER2 can be used to identify objects that are potentially only partially described by this measure. Table 4-3 of Chapter 4 describes how to further examine these identified objects to determine whether or not the OIP1 and OIP2 measures of object dependency are able to fully describe them. Section 4.2.4.6 of Chapter 4 also discusses the

reasons that external function writing feature 8.4.5 is not described by measures. Figure 5-11 shows that measure OER5:OCM4 can be used to identify objects whose measured description of dependency is potentially affected by this loss of measure detail. Table 4-3 of Chapter 4 describes how to further examine the objects identified by measure OER5:OCM4 to determine whether or not they posses the natural language feature 8.4.5 that is not described by measures.

Object Modularity Sub-characteristics	Common Feature	Measure
Interface Dependence : Interface Implementation Dependence: Interface Size	5.2.1	OIS1:OCM1
Interface Dependence : Interface Implementation Dependence: Data Exposure	5.3.1	ODE1:OCM1
Interface Dependence : Interface Implementation Dependence: Interface Size	5.2.3.3	OIS7:OCM2
Interface Dependence : Interface Implementation Dependence: Data Exposure	5.3.2	ODE2:OCM2
Interface Dependence : Interface Implementation Dependence: Interface Size	5.2.3.3	OIS8:OCM3
Interface Dependence : Interface Implementation Dependence: Data Exposure	5.3.2	ODE3:OCM3
External Relationships	6.1.5	OER5:OCM4
Connection Obscurity : Non-standard Connection : Unexpected Relationship	7.4.6	OUER6:OCM4
External Relationships	6.1.6	OER6:OCM5
Connection Obscurity : Non-standard Connection : Unexpected Relationship	7.4.3	OUER3:OCM5
External Relationships	6.1.7	OER7:OCM6
Connection Obscurity : Non-standard Connection : Unexpected Relationship	7.4.4	OUER4:OCM6
External Relationships	6.1.8	OER8:OCM7
Connection Obscurity : Non-standard Connection : Unexpected Relationship	7.4.5	OUER5:OCM7
Connection Obscurity : Unstated Relationship	7.2.1	OUR1:OCM8
Connection Obscurity : Non-standard Connection : Unexpected relationship	7.4.1	OUER1:OCM8
Connection Obscurity : Unstated Relationship	7.2.2	OUR2:OCM9
Dependency : Service Invocation	8.1.1	OSI1:OCM9
Connection Obscurity : Distant Connection	7.3.1	ODC1:OCM10
Connection Obscurity : Non-standard Connection : Unexpected relationship	7.4.2	OUER2:OCM10
Connection Obscurity : Non-standard Connection : Connection via Non-standard Interface	7.5.1	ONI1:OCM10

Table 5-32 Measures quantifying features common to several object modularity sub-characteristics

As mentioned in Section 4.2.4.7 of Chapter 4, some features of the object modularity natural language model are common to several modularity sub-characteristics. The implications of this for data analysis are discussed in Section 4.2.3.6 of Chapter 4. Table 5-32 identifies the measures quantifying these common features, facilitating the removal of common measures where the selected data analysis technique dictates such a procedure.

5.2.5 Level of measurement of modularity measures

The C++ class and object modularity measures developed in this thesis are defined at the ratio level of measurement. For these measures, a value of zero indicates an absence of software features that reduce modularity and values above zero indicate decreasing modularity. There are no negative measured values.

The only exceptions to these rules are measures CIS2 and OIS2, counting the number of hidden attributes in a module, and measures CIS4 and OIS4, counting the number of hidden methods in a module. While these measures are made at the ratio level of measurement, a value of zero indicates an absence of software features that increase modularity, which is the opposite of the other measures. Measures CIS2, CIS4, OIS2 and OIS4 need to be analysed with respect to measures CIS1:CCM1, CIS3, OIS1:OCM1 and OIS3, the number of interface attributes and methods. To obtain a ratio level description class interface size where zero indicates high modularity, the following calculations should be performed. Where a class or object has no methods at all, the interface size is undefined and for analysis purposes, should be set to zero.

class attribute interface size = CIS1:CCM1 / (CIS1:CCM1 + CIS2)
class method interface size = CIS3 / (CIS3 + CIS4)

object attribute interface size = OIS1:OCM1 / (OIS1:OCM1 + OIS2)
object method interface size = OIS3 / (OIS3 + OIS4)

These calculations result in data of a ratio level of measurement where zero indicates high modularity and values increasing from zero indicate decreasing modularity.

5.2.6 Content validation of modularity measures

As discussed in section 5.1.4.2, content validation is appropriate to the set of C++ class and object descriptive measures that are developed in this thesis. Figure 5-4 describes the process of software descriptive measure content validation. The CHARMER diagrams, such as the one illustrated in Figure 5-2, facilitate this process of content validation. Step 1 of the content valuation process described in Figure 5-2 is accomplished in the CHARMER diagram by the specification of links from characteristics and sub-characteristics to the software features that affect them. Step 2 of the content validation is accomplished by examining each of these software features and annotating them with importance and frequency ratings. To perform content validation step 3, a potential user of the measures must examine the features identified in the natural language entity model as affecting the levels of characteristic present. The user must be satisfied that the natural language entity model includes all the features they consider important. If essential features are excluded, then the descriptive measures will not describe these essential features and the user must declare the measures to have insufficient content validity for their intended application. If the user judges the features described by the measures to be adequate, then step 4 of the content validation process can proceed.

To accomplish content validation step 4, a potential user must examine the measures quantifying each software feature. If the measures adequately quantify the features considered by the user to be important and/or frequently occurring, then the set of descriptive measures can be declared to have sufficient content validity. If the potential user decides that the measures do not adequately quantify important or frequently occurring software features, then the measures should be declared to have insufficient content validity. A set of measures with sufficient content validity can be applied to the measurement task. A set of measures with insufficient content validity can either be discarded completely, or augmented with new measure definitions to increase the content validity to acceptable levels.

The following section describes the content validation of a hypothetical software system. The Figure 5-4 content validation process is used and is facilitated by the Figure 5-14 object dependency CHARMER diagram.

5.2.6.1 Example of a content validation

A software engineer is interested in evaluating the dependency sub-characteristic of object modularity. They are particularly interested in the dependency of object modules on state information maintained by software elements external to the object. They are not interested in an object's dependency on interface elements provided by ancestor classes. The following points describe the composition of the software system to be measured.

- The software system to be measured has very few global functions and global variables.
- It does not have friend type relationships between modules.
- One class has a static attribute. This class is not involved in inheritance however it does instantiate 10 objects in the system.
- No objects have attributes appearing in their interfaces.
- Approximately one quarter of system classes are involved in inheritance relationships as parents or children or both.

Step 1 of the Figure 5-4 content validation process is accomplished in the CHARMER diagram of Figure 5-14, which describes the relationships between C++ object dependency sub-characteristics, features and measures.

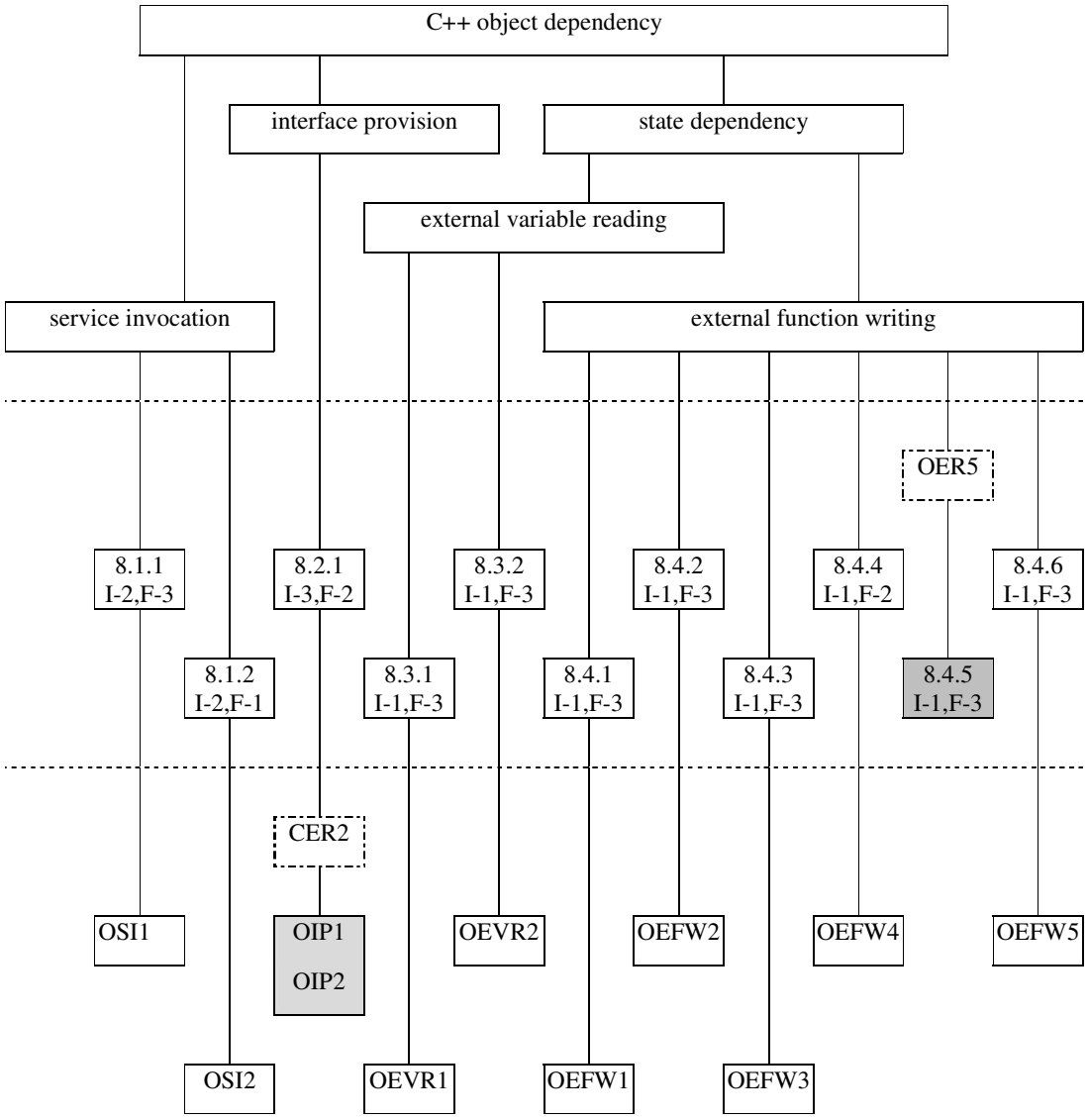


Figure 5-12 Example of content validation of object dependency measures using CHARMER diagram

In step 2 of the content validation process, importance and frequency ratings are assigned to each CHARMER diagram software feature. The importance (I) and frequency (F) ratings used are as follows: 1- high, 2 - medium, 3 - low. In Figure 5-14, the importance and frequency ratings are shown in the natural language feature boxes beneath the feature identification number. The information that the user is interested in measuring object dependency on external state information and is not interested in interface provision dependency is used to assign important ratings to each software feature. Frequency ratings are assigned based on the given

information regarding the composition of the software system to be measured.

In step 3 of the content validation process, the software engineer examines the set of object dependency natural language features and determines that the ones they consider important are included in the set. This means that the theoretical basis from which the measures are developed is appropriate to the description of object dependency the software engineer wishes to obtain by applying the measures.

Step 4 of the content validation process involves the software engineer examining the measures to decide whether or not they adequately quantify the software features. If they do, then the measures can be judged to adequately describe object dependency. In particular, the features identified as being important and/or frequently occurring need to be adequately quantified by measures. The CHARMER diagram indicates that feature 8.2.1 is adequately described by measures OIP1 and OIP2 only under some conditions identified by measure CER2. Measure CER2 is applied to the software system and indicates that 10 classes have distant ancestor classes. Examination of the inheritance hierarchies of these 10 classes shows that none inherit from more than one version of an ancestor class. This means that, for this software system, the mathematical model of object dependency is able to describe all their member elements and that measures OIP1 and OIP2 correctly describe these classes. Hence, feature 8.2.1 is correctly described for all objects in the measured software system.

The CHARMER diagram also indicates that feature 8.4.5 is not described by any measures and that measure OER5:OCM4 indicates whether or not the software system to be measures has this feature. Measure OER5:OCM4 is applied to the software system and indicates that one object-class has a static attribute. Examination of this object-class shows that the static attribute is hidden within the object-class and is directly read from and written to several of the object-class's member methods. In the software system to be measured, only this object-class possesses feature 8.4.5 and hence, the validity of its measured levels of dependency is reduced by the lack of measures to describe feature 8.4.5. For all other system object-classes, the validity of their measured levels of dependency is not affected by the lack of measures to describe feature 8.4.5.

Examination of the Figure 5-14 CHARMER diagram shows that measures are defined to quantify all the remaining important and/or frequently occurring object dependency features. The software engineer judges that the measures adequately describe the dependency of the objects within the system to be measured with the exception of the single object-class with a

hidden static attribute. It is reasonable to declare the set of object dependency measures to have sufficient content validity for this software system because the majority of objects are adequately described by the object dependency measures. Analysis of the measured data obtained from this system should take into account the fact that the object-class with the hidden static attribute, identified by measure OER5:OCM4, has a higher level of external function writing dependency than the measured values indicate.

5.3. Conclusion

The third stage of the systematic measure development process is the development of operational definitions of software characteristics. This is accomplished by defining measures to quantify the software features identified in the natural language entity model as affecting the levels of characteristics present in the software. The prerequisites of the operational definition stage are the conceptual definition of the characteristic to be described by the measures, and the natural language entity model describing the features of the software that affect the level of characteristic present. Together, the conceptual definition and natural language entity model express the theoretical basis from which the measures are developed. An optional prerequisite to the operational definition stage is a mathematical entity model describing the software features identified in the natural language entity model.

A set of natural language measure definitions is the primary product of the operational definition stage. Where a suitable mathematical entity model has been defined in the entity modelling stage of measure development, a set of mathematical measure definitions are a secondary but important product of this stage. The mathematical measure definitions should complement the natural language measure definitions by defining the characteristic of interest in precise, unambiguous terms. Included with each measure definition should be a statement of the level of measurement it achieves. This information will guide a future user in the selection of appropriate analysis techniques to apply to the data obtained from applying the measures to a software system.

Before applying the measures to a system, a user should assess the content validity of the set of measures. Content validity establishes the degree to which the set of measures is appropriate for a proposed application. The theoretical basis from which the measures are developed, as expressed in the conceptual definition and natural language entity model, and the degree to which the measures quantify these natural language model features, must be considered when

determining measure content validity.

Section 5-2 of this chapter demonstrates the process of developing the operational definition of C++ class and object modularity. C++ class and object modularity sub-characteristics are operationally defined in terms of natural language and mathematical measure definitions. The basis for this operational definition is provided by the natural language and mathematical entity models of C++ class and object modularity developed in section 4.2 of Chapter 4. The C++ object mathematical entity models were unable to completely describe all the features of their associated natural language models. These mathematical model shortcomings need to be taken into account when operationally defining object modularity.

The CHARMER diagrams illustrate the operational definition of the identified C++ class and object modularity sub-characteristics. These diagrams describe the measures defined to quantify natural language models features. They also show the natural language model features whose measured description is affected by mathematical model shortcomings and also show features that are not described by measures. The information presented in the CHARMER diagram facilitates the content validation of the modularity measures.

The measurement instrument implementation stage of measure development, as presented in Chapter 6, can proceed once the prerequisite mandatory natural language and optional mathematical measure definitions have been developed to quantify the features of the software that affect the levels of characteristic present.

6. Measurement Instrument Implementation

This chapter describes the measurement instrument implementation stage of software descriptive measure development. This stage corresponds to the shaded boxes in the Figure 6-1 diagrammatic representation of the measure development process. Section 6.1 of this chapter describes the implementation of descriptive measures of software within a software based measurement instrument and section 6.2 demonstrates the implementation of the C++ class and object modularity mathematical measure definitions developed in section 5.2 of Chapter 5.

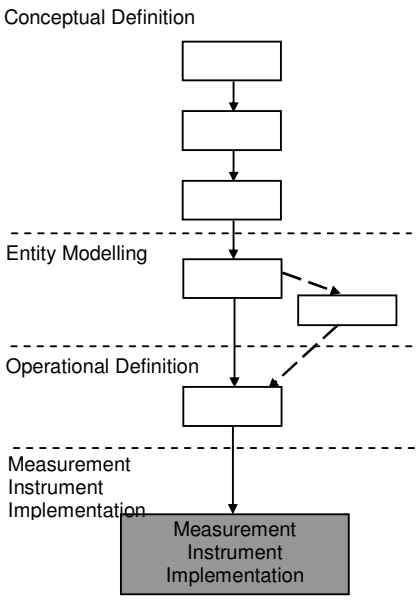


Figure 6-1 Measurement instrument implementation stage of the measure development process

6.1. Stage 4 of measure development process - measurement instrument implementation

In the measurement instrument implementation stage, a tool is developed to collect, from a software entity, data describing the characteristic of interest. In software measurement, it is usual to implement measures within a software based measurement instrument capable of automatically analysing the software system to be measured and producing the measurement data. Abreu, Goulao and Esteves (1995 p.44) note that "the [metrics] collection process is really a repetitive, tedious, boring, time-consuming and expensive task for humans!". A software based measurement instrument can reliably collect measurement data from software system documents if these documents have a regular syntax that can be analysed automatically. Software system source code is an example of a software document from which measurement data can be automatically collected using a software based measurement instrument. A disadvantage of a software based measurement instrument is that it is only able to collect data related to the structure and syntax of the software. This in turn limits the type of measures that can be collected and hence the type of characteristics that can be described by these measures.

The measurement instrument implementation discussed in this chapter is limited to measures that have been formally defined in terms of a mathematical entity model. This represents an ideal case and reflects the C++ class and object modularity implementation presented in section 6.2 of this chapter.

The many different approaches to software measurement "stop at the point at which measurement concepts or specific metrics are identified. They do not define how such measures can be collected and stored, nor (in general) do they define how they can be analyzed." (Kitchenham, Hughes & Linkman 2001, p788). One exception to this is Arisholm, Briand and Foyen (2004) who describe the operation of a tool for collecting coupling data. Limitations of measurement instruments are generally not discussed in detail when software measures are implemented and collected. The limitations of the measurement instrument selected to implement a set of measures can have a significant effect on the quality of the final description of the software obtained. In particular, it may be necessary to tailor measure definitions to fit the capabilities of the selected measurement instrument. The modified measures thus implemented may not provide the same description of the software as the original measure definitions. This potential change of measure validity should be recognised and documented so that it can be taken into account when analysing and interpreting measured data obtained from applying the measurement instrument.

Figure 6-2 illustrates the components of a software based measurement instrument. The elements of this measurement instrument are the source code analyser, the basic software model, the software measurement model, the basic model to measurement model transformations and the software measure definitions.

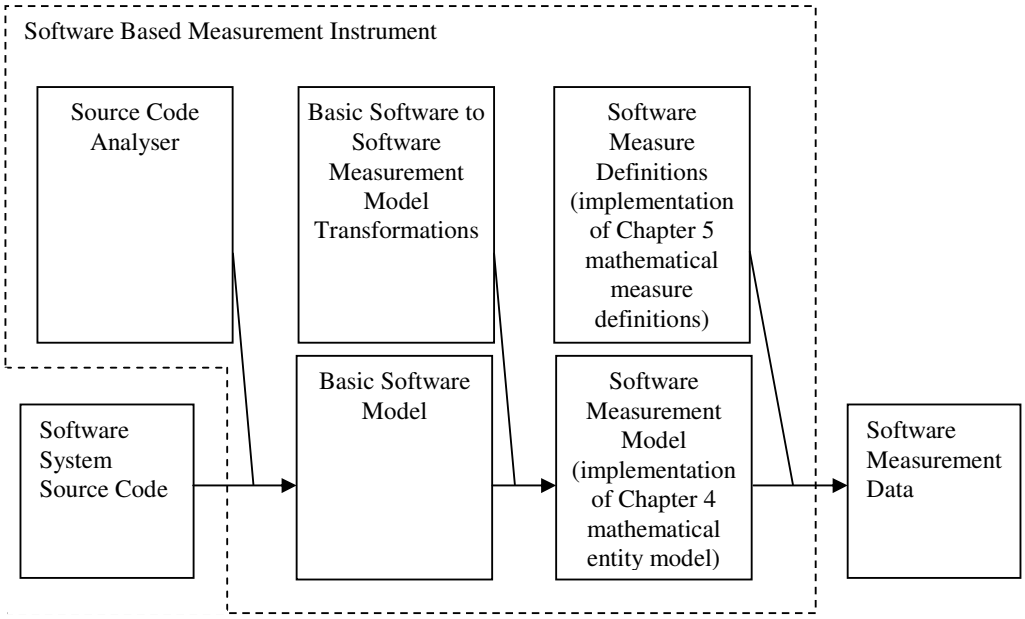


Figure 6-2 Elements of a software based measurement instrument

The input to this type of measurement instrument is the source code of the software system to be measured. The output from this measurement instrument is the software measurement data. The arrows between elements of Figure 6-2 represent transformation processes. These transformations are accomplished by their associated measurement instrument element. The data to fill the basic software model is obtained by parsing the software system source code through a source code analyser. Data manipulations transform the basic software model data into the software measurement model data. This software measurement model represents the implementation of the mathematical entity models defined in the entity modelling stage of measure development. Applying the measure definitions to the software measurement model data generates the software measure data. These measure definitions represent the implementation of the mathematical measure definitions developed in the operational definition stage of measurement development. Measurement instrument limitations, manifest in an inability to implement the full set of defined measures, can be caused by problems implementing one or more of the components of this measurement instrument. As each component is implemented, any problems encountered should be documented so that the impact on the measurement data obtained from the measurement instrument can be assessed.

The following sections describe the measurement instrument implementation stage of measure development in terms of its prerequisites, performance and products. This description is specific to software based measurement instruments implementing measures that have been mathematically defined.

6.1.1 Prerequisites to the measurement instrument implementation stage

A set of measure definitions formally defined in terms of a mathematical entity model is prerequisite to the software based measurement instrument implementation stage of measure development. It is important that these measure definitions specify precisely how each measure is to be made because software based measurement instruments cannot accommodate the implementation of ambiguous measure definitions. Another prerequisite is a measurement instrument capable of implementing the set of measures and performing the measurement operation by extracting the required information from the software system and transforming this into the measured data. If the selected measurement instrument is not capable of implementing the full set of measures, the measure developer must decide whether to seek a different, more capable measurement instrument or accept the limitations of the available instrument and proceed with the implementation.

6.1.2 Performance of the measurement instrument implementation stage

Once the prerequisite requirements have been met, the measurement instrument can be implemented. The Figure 6-2 measurement instrument is implemented progressively from left to right. The first step is to define and implement the basic software model and implement the software document parser that extracts the data needed to fill this model. Following this, the transformations to generate the software measurement model from the basic software model must be defined and implemented. Finally, the mathematical measure definitions are implemented. These definitions transform data from the software measurement model into the software descriptive measurement data.

If the selected measurement instrument is unable to fully implement the mathematical software entity model within the software measurement model then the shortcomings of the software measurement model should be described. Similarly, if any mathematical measure definitions are not fully implemented within the measurement instrument this shortcoming should be described.

As Figure 1-3 from Chapter 1 shows, the level of measurement, validity and reliability of the implemented measures should be assessed, even though the level of measurement and validity of the set of defined measures have been previously assessed as part of the operational definition stage of measure development. This reassessment is necessary because measurement instrument shortcomings may change the level of measurement and validity of the implemented measures. The process of level of measurement and validity assessment of implemented measures is discussed in section 6.1.4. The assessment of software based measurement instrument reliability is discussed in section 6.1.5.

6.1.3 Products of the measurement instrument implementation stage

The primary product of the measurement instrument implementation stage is a software based measurement instrument capable of automatically analysing a software document and transforming the data obtained from this analysis into data describing the levels of the characteristic of interest present in the analysed software system.

If the measurement instrument is able to implement the full set of measures from the operational definition stage of measure development, then the **char**acteristic to **me**asure **r**elationship (CHARMER) diagrams and level of measurement statements from the operational definition stage are applicable to the implemented measurement instrument. If however the measurement instrument is unable to implement the full set of measures defined in the operational definition stage of measure development, then the CHARMER diagrams need to be updated to reflect the measurement instrument shortcomings. It may also be necessary to restate the level of measurement of measures that have been modified in the implementation.

6.1.4 Assessing implemented measure level of measurement and validity

As previously mentioned, before a measurement instrument is applied to a particular measurement task, the level of measurement, validity and reliability of the implemented set of measures should be assessed. A measurement instrument should only be applied when its levels of validity and reliability are judged to be sufficient for an intended application. The level of measurement of the resulting data needs to be known so that appropriate analysis techniques can be applied.

Where measure definitions have been modified to accommodate limitations of the measurement instrument, their level of measurement may have changed. The level of measurement achieved by implemented measures should be reassessed and restated as part of the measurement implementation stage of measure development.

Limitations of the measurement instrument may also result in the reduction of the validity of the set of measures. The measure validity determined at the operational definition stage represents the maximum level of validity that the set of measures can achieve. If the measures can be fully implemented within the measurement instrument, their level of validity will be the same as for the operational definition stage. Any limitations in the ability of the measurement instrument to implement the mathematical entity model as the software measurement model, or the mathematical measure definition as the software measure definitions will reduce the maximum possible validity of the set of measures. A content type validation will be performed on the measures implemented in this thesis. The process of content validation is the same as described in section 5.1.4.2 of Chapter 5.

6.1.5 Reliability of software based measurement instruments

Reliability is the "degree to which an instrument measures the same way each time it is used under the same conditions with the same subjects." (Sproull 1995, p. 74). The validity of a set of measures should be established before determining reliability because a reliable measure that is invalid is not useful.

The five main types of reliability are test-retest reliability, alternative or equivalent forms reliability, split-sample or split-half reliability, internal consistency reliability, and interrater or scorer reliability (Diamantopoulos & Schlegelmilch 1997, p. 36; Sproull 1995, p. 83). Of these, test-retest reliability and interrater reliability are most applicable to software measurement. Test-retest reliability is an estimate of the degree to which a measure produces the same result when administered to the same subject at different times (Sproull 1995, p. 84). Interrater or scorer reliability is an assessment of the degree to which different people "rate the same variables in the same way" (Sproull 1995, p. 89). Interrater reliability is not applicable to measures collected using a software-based instrument as no human judgement is needed when taking the measures from the software. With regard to test-retest reliability, a software based measurement instrument will always produce the same results when applied to the same subject at any time. Thus, the reliability of a software based measurement instrument should always be

considered acceptable.

6.1.6 Practical Considerations

A significant consideration of measurement instrument implementation is how the data needed to fill the basic software model is to be extracted from the software document. While a software based application may be able to automatically parse the software document, it may not be able to extract all the required data. Missing data from the basic software model means that data will also be missing from the software measurement model, which in turn means that some measures cannot be implemented. Another consideration when extracting data from the software document is the way in which it is to be entered into the basic software model. For instance, the software code parsing application may present the data it extracts in a format that is not compatible with the data structures of the basic software model. In this case, a means must be found to modify the format of the data.

Once the basic software model is implemented, the transformations to derive the software measurement model must be defined and implemented. If the selected measurement instrument is unable to perform all the required transformations then measures relying on the information missing from the software measurement model cannot be implemented. The ability of the selected measurement instrument to implement the mathematical measure definitions is an important consideration. Measures defined in a way that is not compatible with the format required by the measurement instrument may need to be redefined or an alternative measurement instrument used.

The size or complexity of a software system to be measured may affect the operation of the measurement instrument. A prototype measurement instrument such as the one demonstrated in section 6.2, constructed from several general purpose applications, may be able to adequately measure relatively small software systems. Problems may arise when an attempt is made to measure a large complex software system. In particular, the measurement instrument may lack the capacity to perform the transformation of basic software model to software measurement model. To overcome problems related to size of the measured system, a large software system may need to be broken down into smaller components that are measured separately. Alternatively, the measures could be implemented on a more powerful measurement instrument.

A final practical consideration is that of human errors made during the implementation stage. For example, a mathematical measure definition could be implemented incorrectly resulting in it generating incorrect measures. A bug in the implementation of a set of measures may not be immediately apparent and could result in an unrecognised reduction in the validity of measured data obtained from a software system. To reduce the chances of this type of error occurring in a measurement instrument, system testing should be undertaken to ensure that each measure is operating correctly.

The remainder of this chapter demonstrates the measurement instrument implementation stage for the implementation of descriptive measures of C++ class and object modularity.

6.2. Implementation of C++ class and object modularity measures

This section describes the implementation of C++ class and object modularity measures within a software based measurement instrument. Figure 6-3 illustrates the measurement instrument used for this implementation.

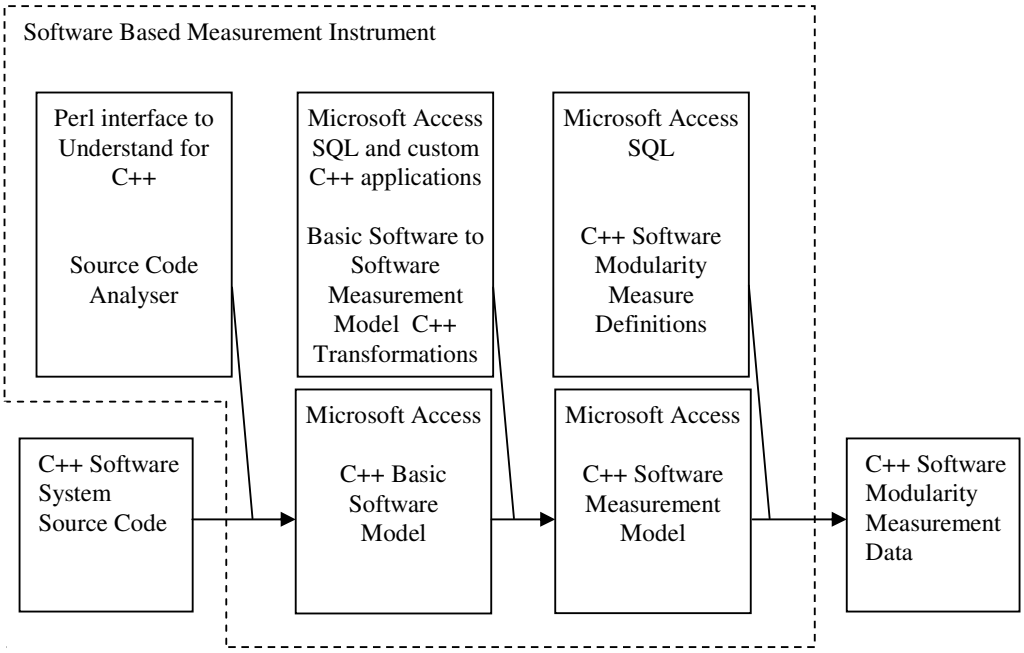


Figure 6-3 Measurement instrument implementing C++ class and object modularity measures

The basic software model describes the elements of the software that can be directly extracted from the software document by a source code analyser. The elements of the basic software model are selected on the basis that it is possible to derive the software measurement model from them. In this implementation, the data to fill the C++ basic software model is extracted from the C++ software system source code using the Understand for C++ source code analyser (Scientific Toolworks Inc. 2003).

Initially, the data is stored within the Understand for C++ internal database. A Perl programming language interface enables custom Perl scripts to extract data from this database. This data is output to text files in a format that can be read directly into the Microsoft Access tables implementing the C++ basic software model.

The information available within the basic software model and the ability of the selected measurement instrument to perform the necessary transformations will affect the degree to which the software measurement model is fully implemented. In this implementation, a combination of Microsoft Access queries, Microsoft Visual Basic routines and custom written C++ applications transform the C++ basic software model into the C++ software measurement model. Appendix 2 describes the basic software model transformations that generate the software measurement model.

The measure definitions specify the data that must be taken from the measurement model in order to describe the characteristic of interest. The C++ class and object modularity measures mathematically defined in Chapter 5 are implemented as Microsoft Access queries on the software measurement model database. The measurement data obtained from these queries can be output to data analysis applications such as Microsoft Excel or the Statistical Package for the Social Sciences (SPSS).

Section 6.2.1 discusses the definition and implementation of a C++ basic software model from which the C++ software modularity measurement model is derived. Section 6.2.2 discusses the transformation of this basic software model into the software measurement model. Section 6.2.3 discusses the implementation of the Chapter 5 C++ class and object modularity measures as queries on the C++ software modularity measurement model database. The level of measurement of the implemented measures and the validity and reliability of the implemented measurement instrument are discussed in section 6.2.4.

6.2.1 C++ basic software model implementation

The basic software model describes features of the C++ software that can be obtained by direct code analysis. The elements of the basic software model are selected on the basis that they can be transformed into the elements of the C++ class and object software measurement models. The basic software model is defined in natural language and mathematical terms.

6.2.1.1 Natural language C++ basic software model

The following points define the C++ basic software model in natural language terms. This natural language model is divided into two sections. The first section describes fundamental software elements and the membership, inheritance, friend and association relationships that must exist before various interactions between these elements can occur. The second section describes interactions between C++ software elements that take place due to these relationships. The C++ basic software mathematical model defined in section 6.2.1.2 describes the features identified in this natural language model. Included with each natural language model point are the names of the associated mathematical model sets describing each feature. A one to one correspondence between natural language feature and mathematical model set indicates that the mathematical model is able to describe the associated feature.

- Software elements and relationships between them
  - A class (C) has member method (MM, M), member attribute (MA, A) and member object (MO, O) elements.
  - Class (C) member attributes (MA, A) are instances of, or pointers to, a C++ primitive data types, declared within the class (C) definition. A primitive data type is one of char, double, float, int, long, short, signed or unsigned.
  - Class (C) member methods (MM, M) are the functions and procedures declared within the class (C) definition.
  - Class (C) member objects (MO, O) are direct instances of, or pointers to instances of (OIC) a class (C).
  - A class (C) is divided into interface and hidden regions. Interface elements have a public or protected level of protection (MM, M, MA, A, MO, O) and hidden elements have a private level of protection (MM, M, MA, A, MO, O).

- The Inherits Parent (IP) type relationship occurs only between classes (C). A child class (C) can inherit from one or more immediate parent classes (C). It can only immediately inherit from one version of a parent class.
- A class (C) may have friend global functions (FF, F), friend classes (FC, C) and friend methods (FM, M).
- A class (C) may have global objects (O) within its scope (SO).
- A class (C) may have global functions (F) within its scope (SF).
- A class (C) may have global variables (V) within its scope (SV).
- Global function (F) associated objects (FIMO, O) are direct instances of, or pointers to instances of (OIC) a class (C).
- Global objects (O) are direct instances of, or pointers to instances of (OIC) a class (C).

- Interactions between software elements
  - A method (M) can directly class-read (MCReadA) a value from an attribute (A) or class-write (MCWriteA) a value to an attribute (A). Class-read and class-write interactions are allowed between methods and attributes when the method and attribute are members of the same class, or when the attribute is a member of an ancestor to the method's member class.
  - A method (M) can directly class-invoke (MCInvM) a method (M). Class-invoke interactions are allowed between methods when they are members of the same class, or when the invoked method is a member of an ancestor to the invoking method's member class.
  - A method (M) can directly global-read (MGReadV) a value from a global variable (V) or directly global-write (MGWriteV) a value to global variable (V).
  - A method (M) can directly global-invoke (MGInvF) a global function (F).
  - A global function (F) can directly global-read (FGReadV) a value from a global variable (V) or global-write (FGWriteV) a value to a global variable (V).
  - A method (M) can directly read, write or invoke access (MOAccessO) an object (O).
  - A global function (F) can directly read, write or invoke access (FOAccessO) an object (O).

6.2.1.2 Mathematical C++ basic software model

The entity-relationship model describing the C++ basic software model is illustrated in Figure 6-4. The set definitions of the entities and relationships of this model are detailed in Appendix 1. As indicated in the C++ basic software natural language model definition, the mathematical model is able to describe all the features of the C++ basic software natural language model.

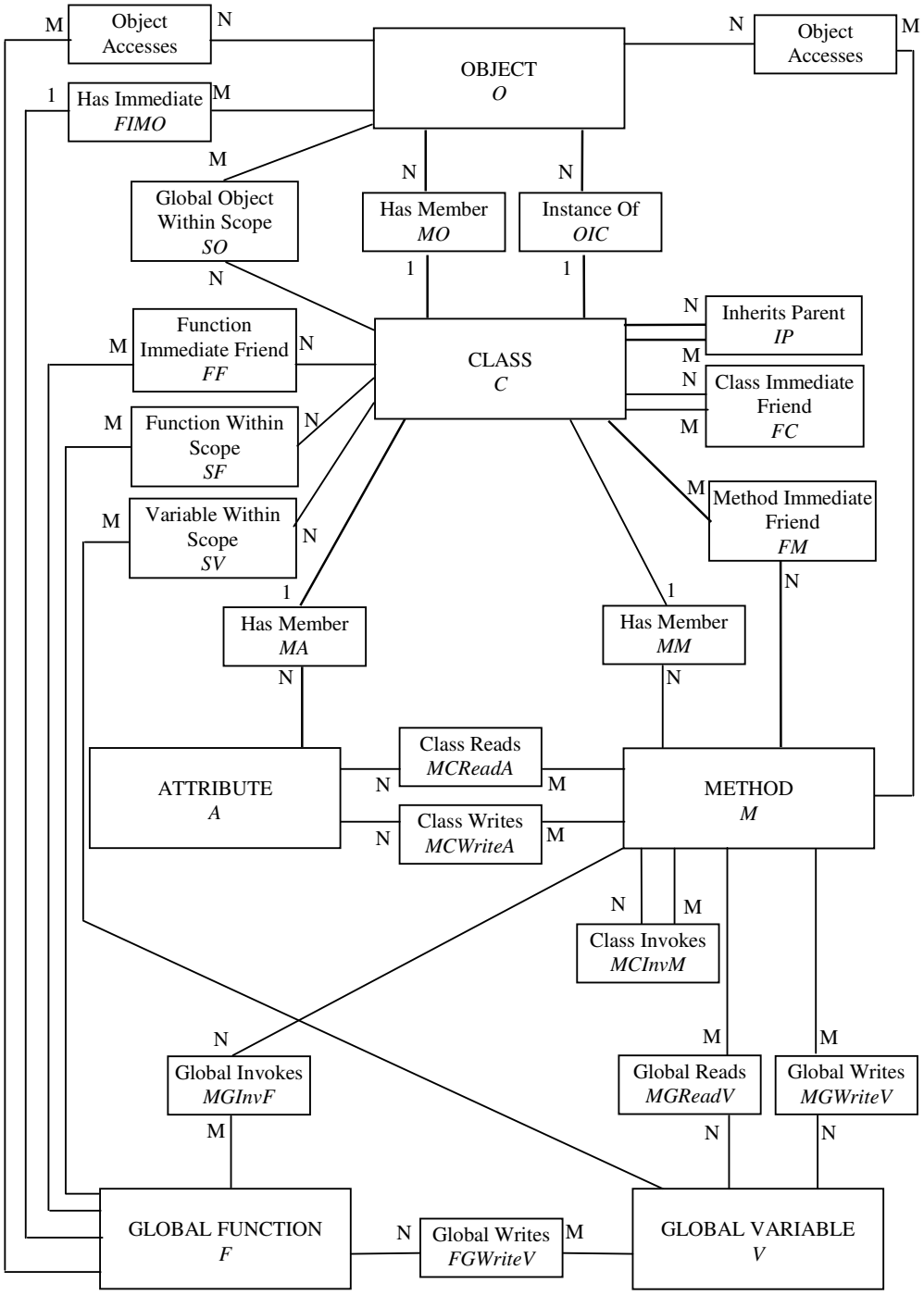


Figure 6-4 Basic software mathematical model

6.2.1.3 Implementation of C++ basic software model

The basic software model is implemented as a Microsoft Access database. The data to fill this database is obtained from the C++ software system source code using the Understand for C++ "reverse engineering, documentation and metrics tool for C and C++ source code" (Scientific Toolworks Inc. 2003). The Understand for C++ application parses C++ software code documents and creates a project file database of software entities and the relationships between them. The required basic software model data is extracted from this project file database using customised Perl scripts developed based on the sample scripts provided by Scitools (Scientific Toolworks Inc. 2003). This data is output to a set of text files corresponding to the entities and relationships of the basic software model. The format of these files is such that they can be read directly into the Microsoft Access basic software model database. Figure 6-7 illustrates this process.

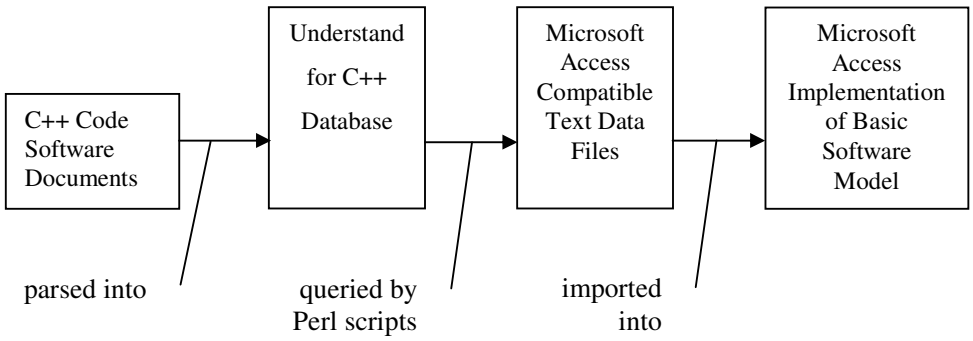


Figure 6-5 Implementation of Basic Software Model

Due to limitations of the Understand for C++ code analysis application, some of the data required by some tables of the basic software model database cannot be extracted from the source code of C++ software systems. This means that the basic software model cannot be fully implemented within the measurement instrument developed in this thesis and described in Figure 6-3. An area for possible future work would be to find and implement a code analysis application capable of extracting all the basic software model data from the C++ source code. The following points describe the elements of the basic software model that cannot be fully implemented in the selected measurement instrument implemented in this thesis.

- In set FOAccessO, the action performed by the global function accessing the object cannot be determined from the Understand for C++ application's code analysis. In Appendix 1, the relation FOAccessO is defined as

  FOAccessO = {(fi, oi, action)} = {(f.fi, o.oi, action) | f ∈ F ∧ o ∈ O ∧ action ∈ {read, write, invoke) ∧ (object o is directly accessed by global function F)}

  Within the measurement instrument implemented within this thesis, the action field of the FOAccessO is assigned an empty string. The revised set definition becomes:

  FOAccessO = {(fi, oi, action)} = {(f.fi, o.oi, "") | f ∈ F ∧ o ∈ O ∧ (object o is directly accessed by global function F)}
- In set MOAccessO, the action performed by the method accessing the object cannot be determined from the Understand for C++ application's code analysis. In Appendix 1, the relation MOAccessO is defined as

  MOAccessO = {(mi, oi, action)} = {(m.mi, o.oi, action) | m ∈ M ∧ o ∈ O ∧ action ∈ {read, write, invoke) ∧ (object o is directly accessed by method M)}

  Within the measurement instrument implemented within this thesis, the action field of the MOAccessO is assigned an empty string. The revised set definition becomes:

  MOAccessO = {(mi, oi, action)} = {(m.mi, o.oi, "") | m ∈ M ∧ o ∈ O ∧ (object o is directly accessed by method M)}
- Set SO cannot be implemented.
- Set SF cannot be implemented
- Set SV cannot be implemented

Due to the inability of the measurement instrument to implement the SO, SF and SV relations and to fully implement the FOAccessO and MOAccessO relations of the basic software model, the description of the software obtained within this implemented basic software model is less detailed than the theoretical basic software entity-relationship model described in section 6.2.1.2. This in turn affects the description obtained when the software measurement model of C++ modularity model is generated from the basic software model. The following section describes the derivation of the software measurement model of C++ modularity.

6.2.2 Software measurement model implementation

The software measurement model is the implementation of the mathematical entity model defined in the entity modelling stage of measure development described in section 4.2 of Chapter 4. The sets of which this model is comprised are defined in Appendix 1. As Figure 6-2 shows,
in a software based measurement instrument, the software measurement model is derived from the basic software model. This derivation is accomplished by transforming the set of the basic software model into the set of the software measurement model.

6.2.2.1 Derivation of C++ software modularity measurement model

Comparison of the C++ basic software mathematical model defined in section 6.2.1.2 and the C++ software modularity measurement models defined in section 4.2 of Chapter 4 shows that most of the sets of the C++ basic software mathematical model also form part of the C++ software modularity measurement model. The exceptions are sets FC (class immediate friend) and FM (method immediate friend). The C++ software modularity measurement model also contains sets that are derived from the C++ basic software model. Table 6-1 lists the derived sets of the C++ software modularity measurement model along with the source C++ basic software model sets from which they are derived. Both source and derived sets are defined in Appendix 1. The transformations that produce the derived C++ software modularity measurement model sets from the source C++ basic software model sets are defined in Appendix 2.

Derived C++ software modularity measurement model sets	Transformation (Appendix 2)	Source C++ basic software model sets
AA - class has accessible attribute	Transformation 1	IP, A
IAA - class has inaccessible attribute	Transformation 1	IP, A
AM - class has accessible method	Transformation 2	IP, M
IAM - class has inaccessible method	Transformation 2	IP, M
AO - class has accessible object	Transformation 3	IP, O
IAO - class has inaccessible object	Transformation 3	IP, O
IMO - class has immediate object	Transformation 4	IP, O, MO,
MICReadA - method indirectly same class reads attribute	Transformation 5	M, A, MCReadA, MCInvM
MICWriteA - method indirectly same class writes attribute	Transformation 6	M, A, MCWriteA, MCInvM
MICInvM - method indirectly same class invokes method	Transformation 7	M, A, MCInvM
MIOCReadA - method indirectly same object class reads attribute	Transformation 8	M, A, IP, MCReadA, MCInvM
MIOCWriteA - method indirectly same object class writes attribute	Transformation 9	M, A, IP, MCWriteA, MCInvM
MIOCInvM - method indirectly same object class invokes method	Transformation 10	M, IP, MCInvM
CEF - class element immediate friend to class	Transformation 11	FC, FM
IDA - class inherits distant ancestor class	Transformation 12	IP
CIF - class element inherited friend to class	Transformation 13	FC, FM, IP
FIF - global function inherited friend to class	Transformation 14	FF, IP

Table 6-1 C++ software modularity measurement model sets derived from C++ basic software model source sets

6.2.2.2 Implementation of C++ software modularity measurement model

Figure 6-3 shows that the C++ software modularity measurement model is implemented within a Microsoft Access relational database. The transformations required to derive this software modularity measurement model from the C++ basic software model are implemented within this database using a combination of Microsoft Access SQL statements, Microsoft Visual Basic routines and custom written C++ programs. The C++ basic software model source sets from which the derived sets are generated are fully implemented within the measurement instrument

as are the transformations needed to generate the C++ software modularity measurement model derived sets. This means that all the derived sets of the C++ software modularity measurement model are fully implemented within the selected measurement instrument.

As previously discussed in section 6.2.1.3, the C++ basic software model sets SO, SF and SV are not implemented and sets FOAccessO and MOAccessO are not fully implemented within the selected measurement instrument due to limitations of the Understand for C++ source code analysis application. The SO set forms part of the C++ object external relationships mathematical model defined in section 4.2.4.3.2 of Chapter 4. It also forms part of the C++ object connection obscurity mathematical model defined in section 4.2.4.4.2 of Chapter 4. The SF and SV sets form part of the C++ class external relationships mathematical model defined in section 4.2.3.3.2 of Chapter 4. The FOAccessO and MOAccessO sets form part of the C++ object connection obscurity mathematical model defined in section 4.2.4.4.2 of Chapter 4 and the C++ object dependency mathematical model defined in section 4.2.4.5.2 of Chapter 4. All these models are not fully implemented within the selected measurement instrument. The table below summarises the C++ software modularity measurement model implementation. It shows that all but the class external relationship, objects external relationship, object connection and object dependency mathematical entity models are fully implemented in the selected measurement instrument.

C++ modularity mathematical entity model (Chapter 4)	Implemented C++ software modularity measurement model
C++ class interface dependence (section 4.2.3.2.2)	Fully implemented within selected measurement instrument.
C++ class external relationships (section 4.2.3.3.2)	Unable to implement SF and SV sets of this model.
C++ class connection obscurity (section 4.2.3.4.2)	Fully implemented within selected measurement instrument.
C++ class dependency (section 4.2.3.5.2)	Fully implemented within selected measurement instrument.
C++ object interface dependence (section 4.2.4.2.2)	Fully implemented within selected measurement instrument.
C++ object external relationships (section 4.2.4.3.2)	Unable to implement SO set of this model.
C++ object connection obscurity (section 4.2.4.4.2)	Unable to implement SO set of this model. Unable to fully implement FOAccessO and MOAccessO sets of this model.
C++ object dependency (section 4.2.4.5.2)	Unable to fully implement FOAccessO and MOAccessO sets of this model.

Table 6-2 Implementation of C++ mathematical entity models as C++ software modularity measurement model

The inability of the measurement instrument to fully implement sets SF, SV, SO, FOAccessO and MOAccessO affects its ability to fully implement some of the mathematical modularity measures defined in section 5.2 of Chapter 5. The following section describes measure implementation within the selected measurement instrument Microsoft Access database.

6.2.3 Software modularity measures implementation

Figure 6-2 shows that the software measure definitions component of the measurement instrument are the implementation of the mathematical measure definitions developed in the operational definition stage of descriptive measure development. Figure 6-3 shows that the selected measurement instrument implements the measures of C++ class and object modularity defined in section 5.2 of Chapter 5, as Microsoft Access SQL statements within the C++ software modularity measurement model database.

Some measure definitions represent simple query statements while others need to be implemented with a succession of queries. The following examples show how both a simple and a complex measure definition convert to Microsoft Access queries. The underlined statements identify the software modularity measurement model sets used in the query. Joins between these sets are shown in bold print.

Example 1: Simple Query

Measure CUER2 - For each class, the number of other classes that are immediate full or partial friends to the class.

CUER2 = {(ci, total)} = {(x.ci, total) | $\underline{x \in C}$ $\wedge$ total = #{y.friend_ci | $\underline{y \in CEF}$ $\wedge$ **x.ci = y.ci**}}

This measure is implemented within the Microsoft Access software modularity measurement model database as the following query.

```
SELECT C.ci, Count(CEF.friend_ci) AS total
FROM C LEFT JOIN CEF ON C.ci = CEF.ci
GROUP BY C.ci;
```

Example 2: Complex Query

Measure CIS7 is a relatively complex query that has to be implemented in stages within Microsoft Access.

Measure CIS7 - For each class, for each non-constructor or non-destructor interface method, the total number of same class attributes directly read.

CIS7 = {(ci, mi, total)} = {(y.ci, y.mi, total) |y ∈ M ∧ y.protection ∈ {public, protected} ∧ y.purpose ∉ {constructor, destructor} ∧ total = #{z.ai | z ∈ A ∧ **z.ci = y.ci** ∧ (**y.mi, z.ai**) ∈ MCReadA}}

The first stage of the query generates the set 'sub CIS7' = {(ci, mi, ai)} of a class identifier, same class member method and same class member attribute, where the method directly reads from the attribute.

sub_CIS7:

```
SELECT M.ci, M.mi, A.ai
FROM A INNER JOIN (M INNER JOIN MCReadA ON M.mi = MCReadA.mi) ON
(M.ci = A.ci) AND (A.ai = MCReadA.ai)
GROUP BY M.ci, M.mi, A.ai;
```

The second stage identifies all the non-constructor and non-destructor interface class member methods and counts the number of times each of these methods appears in the previously derived set 'sub_CIS7'. This is a count of the number of attributes read by each class non-constructor or destructor method.

```
SELECT M.ci, M.mi, Count([sub_CIS7].ai) AS total
FROM M LEFT JOIN [sub CIS7] ON M.mi = [sub_CIS7].mi
GROUP BY M.ci, M.mi, M.protection, M.purpose
HAVING (((M.protection)="public" Or (M.protection)="protected") AND
((M.purpose)<>"destructor" And (M.purpose)<>"constructor"));
```

In a similar way to these two examples, the defined measures of modularity are implemented within the Microsoft Access software modularity measurement database.

As mentioned in section 6.2.2.2, the selected measurement instrument is unable to implement sets SF, SV and SO and is unable to fully implement sets FOAccessO and MOAccessO, which means that the class external relationships, object external relationships, object connection obscurity and object dependency mathematical entity models are not fully implemented. The measurement instrument is thus unable to fully implement measures of C++ class external relationships, C++ object external relationships, C++ object connection obscurity and C++ object dependency that rely on data contained within these sets. Table 6-2 lists the measures omitted from the selected measurement instrument implementation due to their dependence on the SF, SV, SO, FOAccessO or MOAccessO sets.

Modularity sub-characteristic	Omitted Measure	Set
Class External Relationships : **Within the System**	CER8	SF
	CER9	SV
Object External Relationships : **Within the System**	OER2	SO
Object Connection Obscurity : Non-standard Connection : **Connection via Non-standard Interface**	ONI2 and ONI3	MOAccessO
	ONI4	FOAccessO
Object Connection Obscurity : **Variable Connection**	OVC2	SO
Object Dependency : **Service Invocation**	OSI2	MOAccessO
Object Dependency : State Dependency : **External Variable Reading**	OEVR2	MOAccessO
Object Dependency : State Dependency : **External Function Writing**	OEFW5	MOAccessO
	OEFW3	FOAccessO

Table 6-3 Measures of C++ object modularity omitted from measurement instrument implementation

The CHARMER diagrams of C++ class interface dependence, connection obscurity and dependency defined in sections 5.2.31, 5.2.3.3 and 5.2.3.4 respectively remain unchanged for the selected measurement instrument implementation. The CHARMER diagrams of C++ object interface dependence and external relationships defined in sections 5.2.4.1 and 5.2.4.2 respectively also remain unchanged for the selected measurement instrument implementation. The Figure 5.6 CHARMER diagram of class external relationships defined in section 5.2.3.2, Figure 5-11 and Figure 5-12 CHARMER diagrams of object connection obscurity defined in section 5.2.4.3 and the Figure 5-13 CHARMER diagram of object dependency defined in section 5.2.4.4 must be updated to reflect the implemented measures.

6.2.3.1 Class external relationships measure implementation

As Table 6-3 shows, measures CER8 and CER9 are not implemented within the selected measurement instrument. The Figure 5-6 CHARMER diagram describing the measurement of class external relationships needs to be updated. Figure 6-6 is the updated version of Figure 5-6, reflecting the inability of the measurement instrument to implement measures CER8 and CER9. Class external relationship natural language model points 2.1.8 and 2.1.9 are not described by implemented measures. When the set of implemented modularity measures are validated for a specific measurement situation, the potential user will need to decide whether or not the remaining implemented measures provide an adequate description of the software. As Figure 6-6 indicates, there is no means provided by which classes affected by the lack of description of points 2.1.8 and 2.1.9 can be identified. When performing the content validation, it must be assumed that the description of external relationships of all the software system classes is affected by the failure to fully implement the defined CER8 and CER9 measures within the selected measurement instrument.

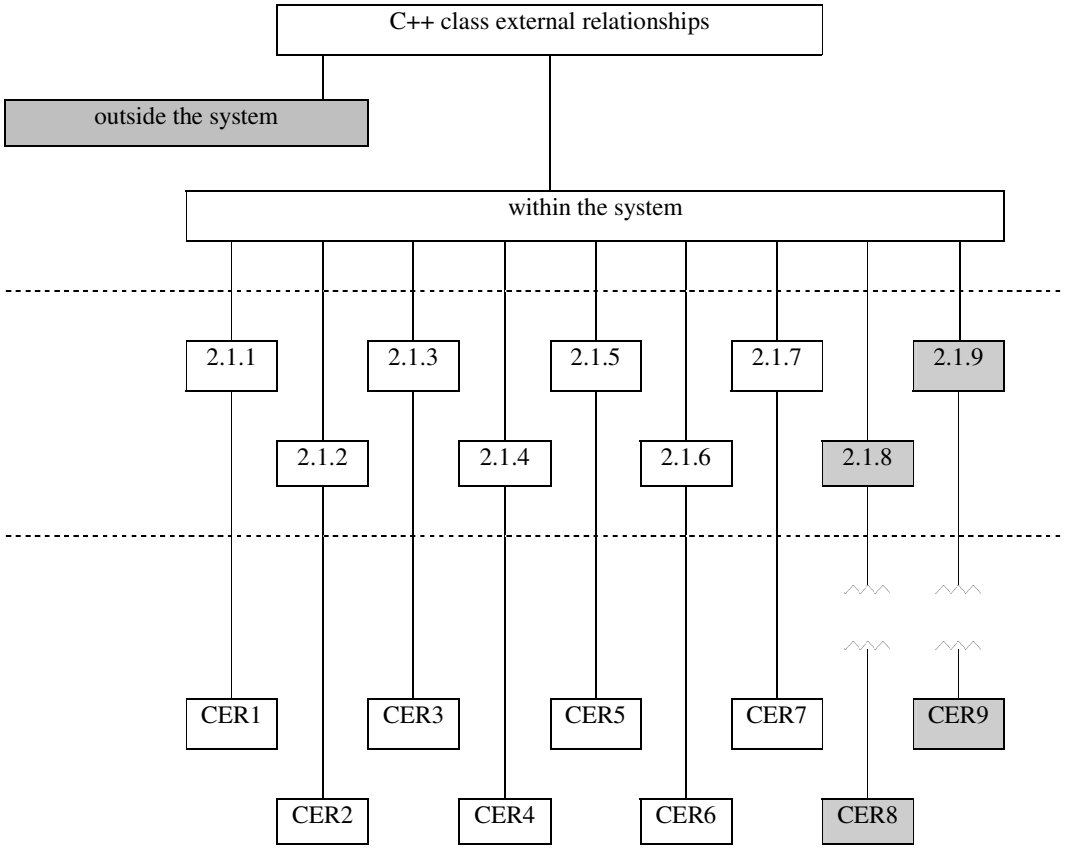


Figure 6-6 Update of Figure 5-6 C++ class external relationships CHARMER diagram, reflecting measurement instrument implementation

6.2.3.2 Object external relationships measure implementation

As Table 6-3 shows, measure OER2 is not implemented within the selected measurement instrument. The Figure 5-10 CHARMER diagram describing the measurement of object external relationships needs to be updated. Figure 6-7 is the updated version of Figure 5-10, reflecting the inability of the measurement instrument to implement measure OER2. Object external relationship natural language model point 6.1.2 is not described by implemented measures. When the set of implemented modularity measures are validated for a specific measurement situation, the potential user will need to decide whether or not the remaining implemented measures provide an adequate description of the software.

As Figure 6-7 indicates, there is no means provided by which classes affected by the lack of description of point 6.1.2 can be identified. When performing the content validation, it must be assumed that the description of external relationships of all the software system object-classes is affected by the failure to fully implement the defined OER2 measure within the selected measurement instrument.

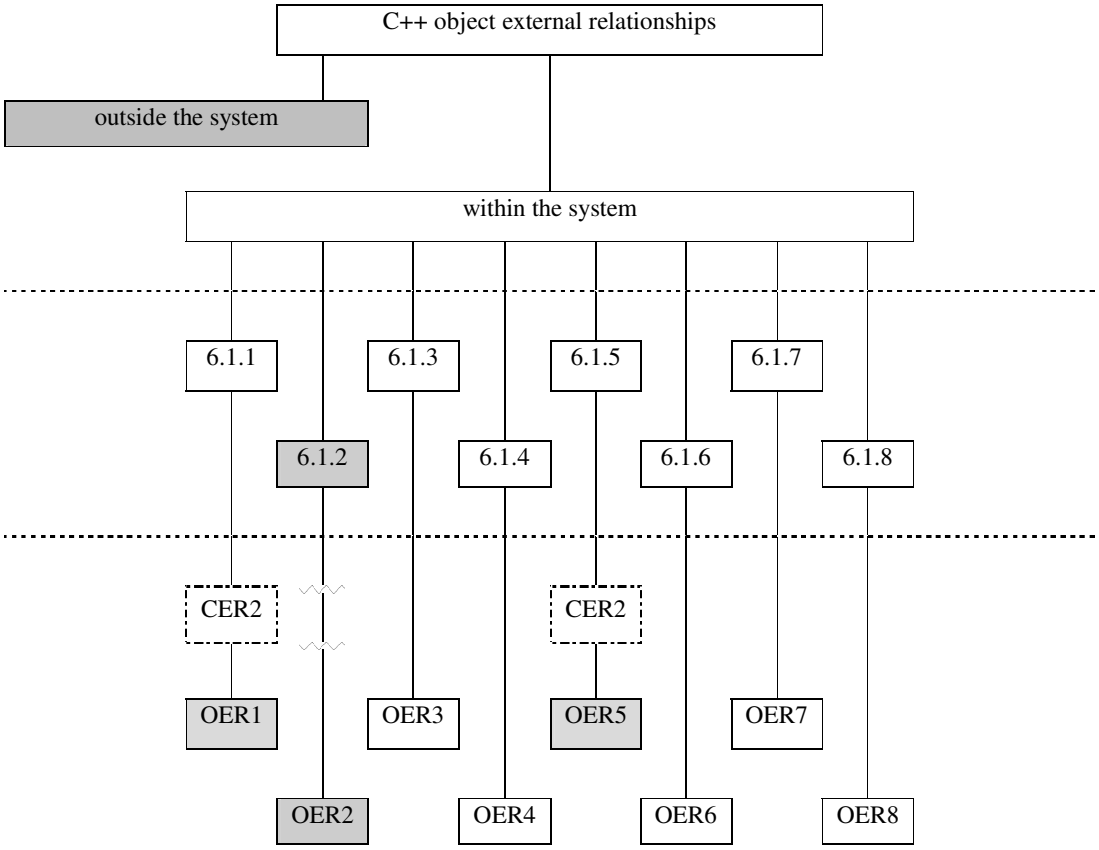


Figure 6-7 Update of Figure 5-10 C++ object external relationships CHARMER diagram, reflecting measurement instrument implementation

6.2.3.3 Object connection obscurity measure implementation

As Table 6-3 shows, measures OVC2, ONI2, ONI3 and ONI4 are not implemented within the selected measurement instrument. Both the Figure 5-11 CHARMER diagram and the Figure 5-12 CHARMER diagram need to be updated. Figure 6-8 is the updated version of Figure 5-11, reflecting the inability of the measurement instrument to implement measure OVC2. Figure 6-9 is the updated version of Figure 5-12, reflecting the inability of the measurement instrument to implement measures ONI2, ONI3 and ONI4. Object connection obscurity natural language model points 7.1.2, 7.5.2, 7.5.5 and 7.5.6 are not described by implemented measures. As Figure 6.8 shows, in some circumstances, the variable connection sub-characteristics of connection obscurity is not described by implemented measures. Figure 6-9 shows that the connection via non-standard interface sub-characteristic of connection obscurity is only

described by a single measure. When the set of implemented modularity measures are validated for a specific measurement situation, the potential user will need to decide whether or not this single measure provides an adequate description of the software.

As Figures 6-8 and 6-9 indicate, there is no means provided by which objects affected by the lack of description of points 7.1.2, 7.5.2, 7.5.5 and 7.5.6 can be identified. When performing the content validation, it must be assumed that the description of connection via non-standard interface of all the software system objects is affected by the failure to fully implement the defined OVC2, ONI2, ONI3 and ONI4 measures within the selected measurement instrument.

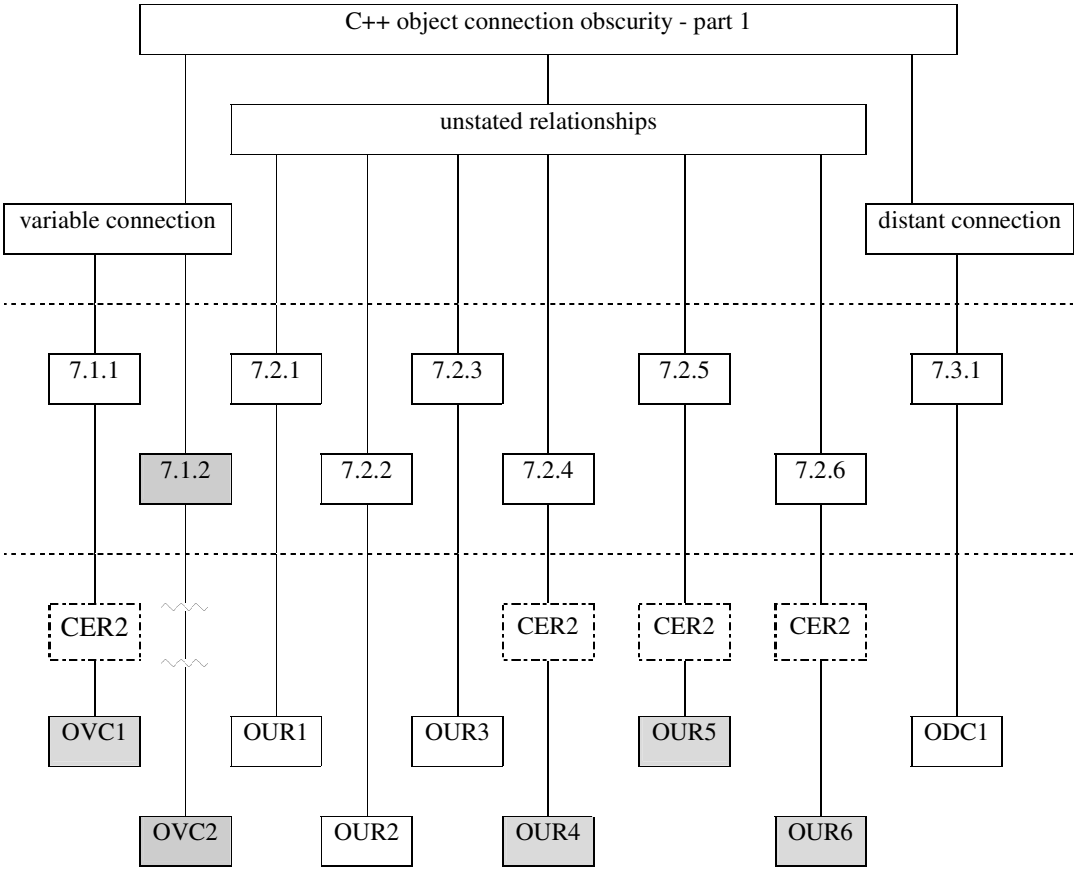


Figure 6-8 Update of Figure 5-11 C++ object connection obscurity CHARMER diagram, reflecting measurement instrument implementation

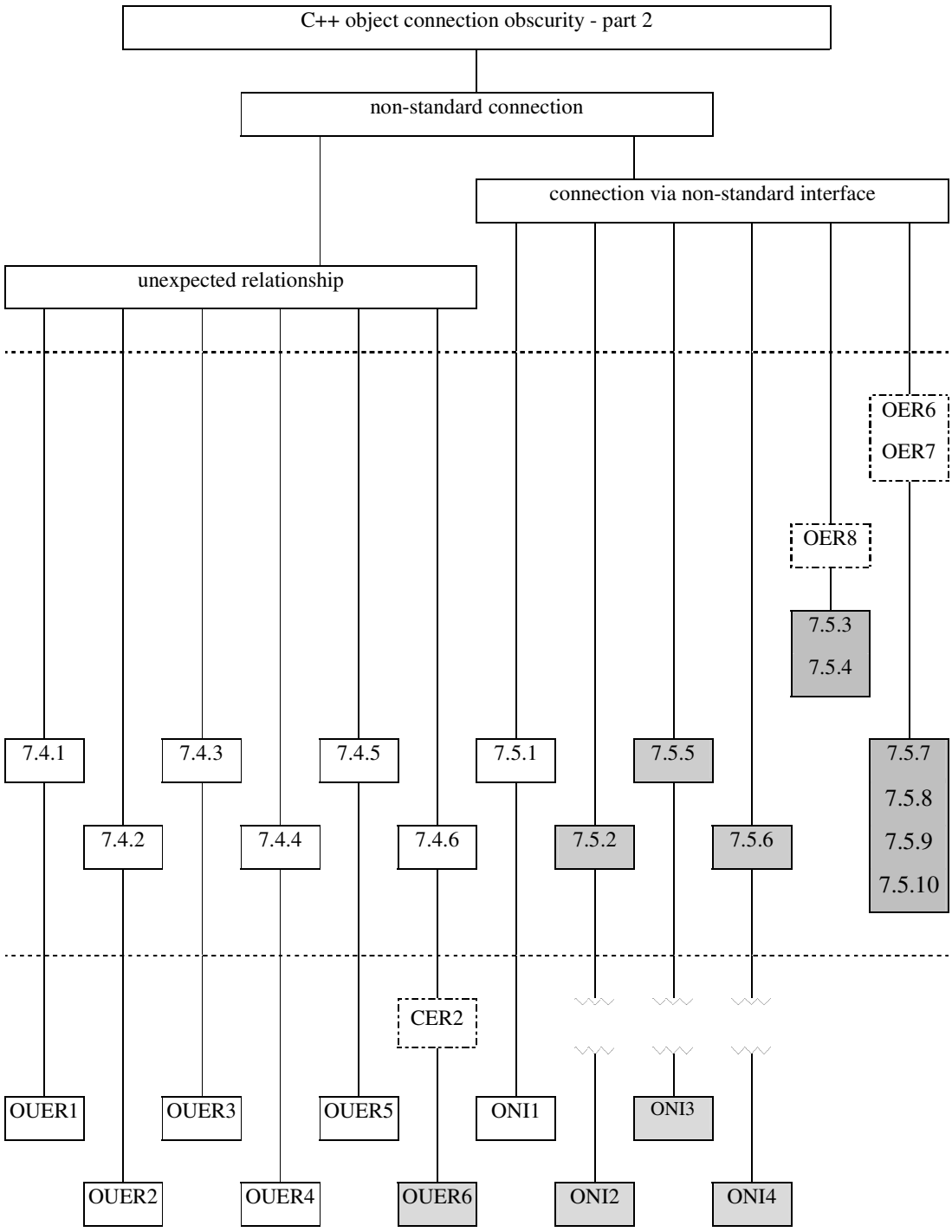


Figure 6-9 Update of Figure 5-12 C++ object connection obscurity CHARMER diagram, reflecting measurement instrument implementation

6.2.3.4 Object dependency measure implementation

Table 6-3 shows that object dependency measures OSI2, OEVR2, OEFW3 and OEFW5 are not implemented within the selected measurement instrument. Figure 6-10 is the updated version of the Figure 5-13 object dependency CHARMER diagram. Object dependency natural language model points 8.1.2, 8.3.2, 8.4.3 and 8.4.6 are not described by implemented measures.

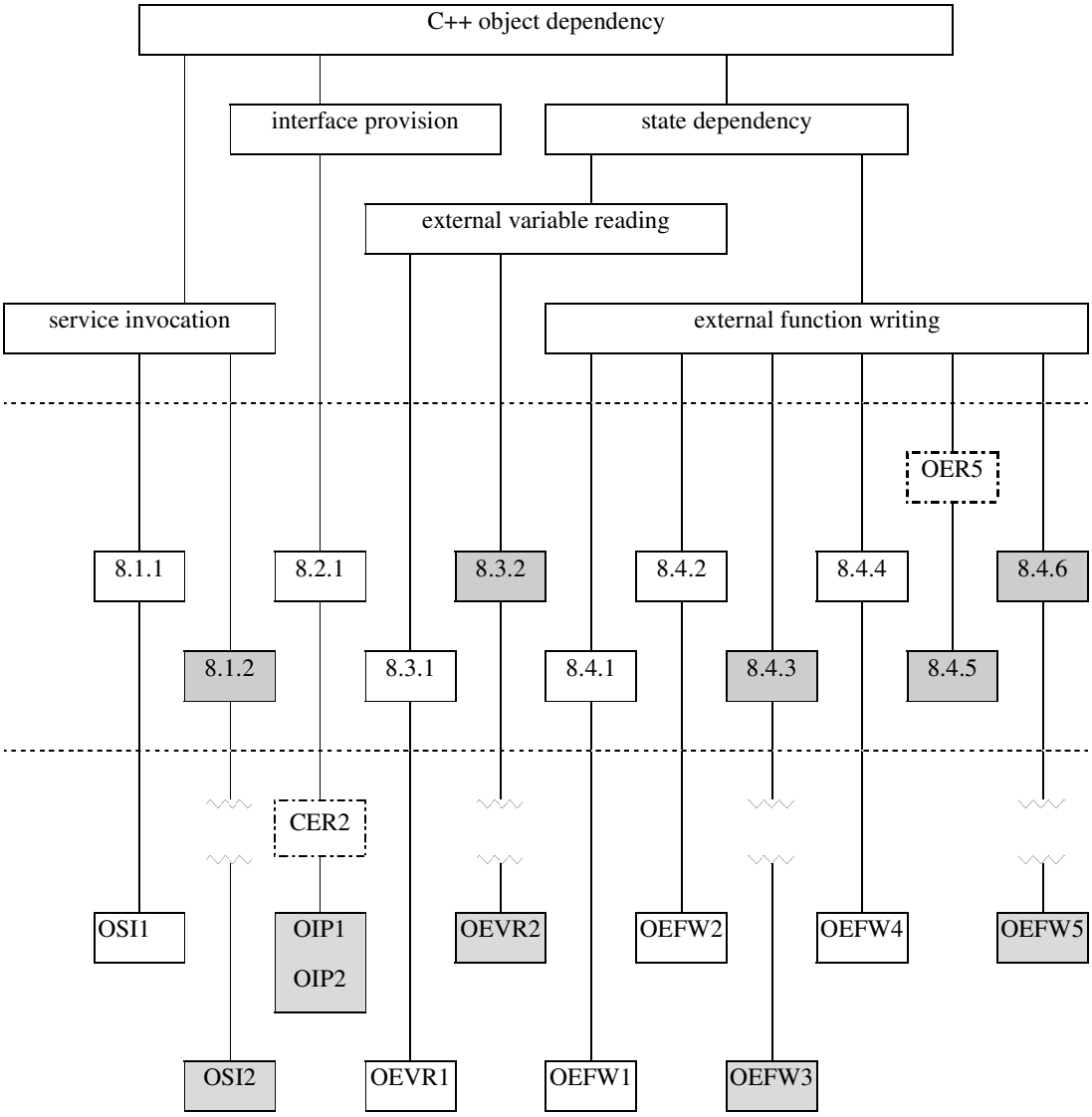


Figure 6-10 Update of Figure 5-13 C++ object dependency CHARMER diagram, reflecting measurement instrument implementation

As Figure 6-10 indicates, there is no means provided by which objects affected by the lack of description of points 8.1.2, 8.3.2, 8.4.3 and 8.4.6 can be identified. When performing the content validation, it must be assumed that the description of dependency of all the software system objects is affected by the failure to fully implement the defined OSI2, OEVR2, OEFW3 and OEFW5 measures within the selected measurement instrument.

The following section discusses the level of measurement and validity of the measures implemented within the selected measurement instrument and, in Chapter 7 a case study is performed that demonstrates the content validation of the implemented measures.

6.2.4 Level of measurement and content validity of implemented measures

The inability of a measurement instrument to implement the full set of defined measures of a software characteristic may be manifest in a change to the level of measurement of some implemented measures and in a change to the content validity of the entire set of implemented measures.

6.2.4.1 Level of measurement

Implementation has not changed the level of measurement of implemented measures from that described in section 5.2.5 of Chapter 5. This means that all the measures are made on the ratio scale and can be analysed using both non-parametric and parametric statistics [Diamantopoulos97] p27.

6.2.4.2 Content validity

Software descriptive measure validation is discussed in section 5.1.4.2 of Chapter 5 and content validity is selected as being appropriate to the measures developed in this thesis. Section 5.2.6 describes a process of software descriptive measure content validation using the CHARMER diagrams. This same process is applicable to the content validation of implemented software descriptive measures. It is important to use CHARMER diagrams that have been updated to reflect the measures implemented within the selected measurement instrument.

Table 6-4 lists the CHARMER diagrams appropriate to the content validation of the measures implemented within the measurement instrument described by Figure 6-3.

C++ modularity sub-characteristic	Measure implementation CHARMER diagram
class interface dependence	Figure 5-5 of Chapter 5
class external relationships	Figure 6-6 of Chapter 6
class connection obscurity	Figure 5-7 of Chapter 5
class dependency	Figure 5-8 of Chapter 5
object interface dependence	Figure 5-9 of Chapter 5
object external relationships	Figure 6-7 of Chapter 6
object connection obscurity	Figures 6-8 and 6-9 of Chapter 6
object dependency	Figure 6-10 of Chapter 6

Table 6-4 Characteristic to measure relationship (CHARMER) diagrams describing implemented measures

Chapter 7 presents a case study in which the implemented measurement instrument is applied to the task of describing the class and object modularity of a small C++ software system. The content validation of the implemented measures using the CHARMER diagrams listed in the above table forms part of this case study.

6.3. Conclusion

Measurement instrument implementation of operational measure definitions is the fourth and final stage of the systematic measure development process. For a software based measurement instrument, such as the one developed in this thesis, a prerequisite of the measurement instrument implementation stage is a set of measures defined in terms of a mathematical entity model of the software. Another prerequisite is a measurement instrument capable of implementing these measures and automatically collecting them from a software system.

A software based measurement instrument capable of analysing a software system and outputting a description of this system in the form of measured data is the primary product of the measurement instrument implementation stage. The other important products of this stage are a statement of the level of measurement achieved by each implemented measure and a set of CHARMER diagrams. These diagrams describe which features of the software natural language models are described by implemented measures, and which are not. The CHARMER diagrams facilitate the content validation of the set of implemented measures.

Section 6-2 of this chapter demonstrates the measurement instrument implementation of the C++ class and object modularity measures defined in section 5.2 of Chapter 5. The measurement instrument selected for this implementation was unable to fully implement the C++ object connection obscurity and dependency mathematical entity models. This in turn meant that several measures of object connection obscurity and dependency could not be implemented. The CHARMER diagrams must be updated to reflect any shortcomings of the measurement instrument so that a content validation can be accurately performed.

Implementation of the software measures within a measurement instrument marks the completion of the systematic process of software descriptive measure development. Chapter 7 describes a case study in which the measurement instrument described in section 6.2 is applied to the task of describing the class and object modularity of a small C++ software system. This case study will show how the developed measures can be validated and applied and the resulting measurement data analysed and interpreted to provide an adequate description of the system's modularity.

7. Application of C++ Class and Object Modularity Measures

The previous chapter described the final stage of the systematic measure development process, one of the products of which is a measurement instrument capable of automatically collecting the measurement data from the source code of a C++ software system. This chapter demonstrates measure validation and the analysis and interpretation of measurement data obtained by applying this measurement instrument to a small C++ software system.

Section 7.1 of this chapter describes an analysis technique involving the calculation of modularity aggregate values. The values describe, for individual classes and object-classes within a measured software system, the relative levels of modularity and modularity sub-characteristics present. Another data analysis technique, based on the calculation of Euclidean distance to describe dissimilarity between pairs of measured classes or object-classes, is also presented.

Section 7.2 describes a content type validation, with respect to the source code of the eMulePlus file sharing client system, of the implemented measurement instrument described in Chapter 6. The eMulePlus system will also be used in the example of construct validation and the two case studies that are also included in Chapter 7. The eMulePlus system was selected because its complete source code was available which meant that all the connections between the system class and objects were available to be measured. It was also selected because it is large enough to provide a significant amount of modularity data and yet small enough to be able to be measured by the prototype measurement instrument developed for this study. The specifications of the eMulePlus system are as follows.

System: eMulePlus file sharing client
Version: eMulePlus-1f
Available: http:\\www.sourceforge/net/projects/emuleplus/
Date: 9th June 2003

The measured eMulePlus system consists of:

- 93823 lines of code
- 24378 lines of comment
- 249 classes
- 4373 class methods
- 509 class attributes
- 278 global functions
- 16 global variables

Section 7.3 describes an example of a construct type validation. The eMulePlus class unweighted and weighted modularity aggregate and Euclidean distance values calculated according to the methods described in section 7.1 are used in this validation. Only class modularity construct validation is performed as the content validation described in section 7.2 indicates that the implemented measures are unable to provide an adequate description of eMulePlus object modularity.

Section 7.4 presented two case studies. The first is a case study of the eMulePlus software system using the unweighted modularity aggregate calculated according to the method of section 7.1 to identify classes and objects with relatively low modularity. The second case study analysing and interpreting the measured data describing the class and object interface dependence of a selected eMulePlus class that was indicated in the previous case study as having low modularity.

In presenting these different examples of validation, analysis and data interpretation, the aim is not to show how the modularity measures should be used, but rather to demonstrate how they can be used and to show by example how the systematic process by which they were developed supports their practical application.

7.1. Data Reduction Techniques

A major challenge in analysing and interpreting the modularity measurement data obtained from measuring a software system is the amount of data that must be presented. Measuring the modularity of the relatively small eMulePlus system resulted in more than 25,000 individual points of data. It is not possible for a person to interpret this amount of raw data and gain an understanding of the levels of modularity present in the system. Some analysis of the measurement data must be undertaken, and the results of this analysis presented in a way that a user can understand. "Typically, customers are not experts in software engineering; they want a "big picture" of what software is like, not a large vector of measures of different aspects." (Pfleeger, Jeffery, Curtis & Kitchenham 1997, p. 41) This section describes two different methods of data analysis that result in a single number that describes the general modularity of individual classes and object-classes. The first method aggregates the various measures of each identified modularity sub-characteristic to obtain, for each measured class and object-class, a single numerical aggregate of modularity. The second method combines all the measured modularity values to calculate a single value of dissimilarity between each pair of classes or object-classes in the measured system.

7.1.1 Calculation of an Aggregate of Modularity

As explained in section 5.2.2 of Chapter 5, the modularity measures have the property that a value of zero indicates an absence of software features that reduce modularity and values above zero indicate decreasing modularity. This property facilitates the calculation of an aggregation of modularity measures for each measured class and object within a software system. Associate Professor Peter Petocz, currently of Macquarie University, Sydney (ppetocz@efs.mq.edu.au) first suggested the calculation of an index as a possible technique for data analysis. From this suggestion, the modularity aggregation process was developed. The following steps describe the calculation of class and object modularity aggregates.

7.1.1.1 Preliminary calculations

1. Calculate
   class attribute interface size = CIS1:CCM1 / (CIS1:CCM1 + CIS2)
   class method interface size = CIS3 / (CIS3 + CIS4)
   object attribute interface size = OIS1:OCM1 / (OIS1:OCM1 + OIS2)
   object method interface size = OIS3 / (OIS3 + OIS4)
   Use these calculated values rather than the CIS1-4 and OIS1-4 to calculate the modularity indices.

   Measures CIEI1 to CIEI8 and OIEI1 to OIEI8 are made on a per method or per attribute basis rather than on a per class or object-class basis. For each class or object-class calculate the median of each of these measures for their member methods or attributes. The median is selected as it is a robust statistic most suited to describing the central location of a skewed distribution. Preliminary examination of the eMulePlus measures of CIEI1 to CIEI8 show that these distributions are strongly positively skewed. For distributions that are not skewed, the median is still a suitable statistic for describing central tendency and so, may be generally used when calculating the modularity aggregate. Alternatively, for normal distributions, the mean can be used instead of the median.

   Use these calculated median values, rather than measures CIEI1 to CIEI8 and OIEI1 to OIEI8, to calculate the class and object modularity indices.

2. Replace all missing values with 0 since missing values give too much emphasis to the values that are there for the case. 0 is the replacement value since it does not reduce the modularity value of the case.

3. Add a Zero class to the set of measured data. All measured values for this class are set to zero. This provides a point of maximum modularity with which to interpret the modularity aggregates of the actual system classes and objects.

The method of calculating the modularity aggregate is different for this artificial Zero class than for the actual measured classes and objects. As recognised by one of the anonymous assessors of this thesis (Anonymous Assessor, 2005), including the Zero class in the general normalisation calculation may have the effect of reducing the range of the normalised measured values by pushing them towards the positive end of the range while the Zero class occurs at the extreme negative end of the range. This effect will be most pronounced when the measured values are much greater than 0. To avoid this situation, the actual measured values will be normalised and the mean and standard deviation from this normalisation will then be used to calculate the new value of the artificial Zero class.

The method to calculate the modularity aggregates is described below. The terminology used is that:

- Major modularity sub-characteristics are those directly related to modularity. They are interface dependence, external relationships, connection obscurity and dependency.
- Minor modularity sub-characteristics are those that are directly associated with each major sub-characteristic. They are interface element interdependence, interface implementation dependence, external relationships within the system, external relationships outside the system, variable connection, unstated relationship, distant connection, non-standard connection, service invocation, interface provision, and state dependency.
- Sub-minor modularity sub-characteristics are directly related to some minor modularity characteristics. They interface size, data exposure, unexpected relationship, connection via non-standard interface, external variable reading and external function writing.

This method of aggregate calculation is based on the idea that each sub-minor sub-characteristic contributes equally to the aggregate value of its associated minor sub-characteristic, that each minor sub-characteristic contributes equally to the aggregate value of its associated major sub-characteristic and that each major sub-characteristic contributes equally to the main characteristic aggregate value.

7.1.1.2 Class and object modularity aggregate calculation

1. Calculate the standard normal distribution Z_X of each modularity measure X except the Zero class and object-class. The distribution Z_X has a mean of zero and a standard deviation of 1. Using a normalised distribution of measurement data ensures that each measure, regardless of its range, contributes equally to each aggregate. The standard normal distributions are calculated as follows (Swift 2001, p. 483).

   $Z\_X = (X - \mu_X) / \sigma_X$

   where μ_X is the mean of distribution X and σ_X is its standard deviation.

2. Using the previously calculated values of μ and σ, calculate the normalised value of the Zero class or object-class for measure X.

   $Z\_X_{Zero} = (x_{Zero} - \mu_X) / \sigma_X$

3. For each minor and sub-minor modularity sub-characteristic for which measures have been directly defined, for each class (i = 1..N+1) and each object class (i = 1..N+1), including the Zero class and object class, calculate the aggregate of the normalised measured values where N is the number of measured classes and the number of measured object classes.

Class Interface Dependence

$A\_CIEI(c_i) = CIEI1(c_i) + CIEI2(c_i) + CIEI3(c_i) + CIEI4(c_i) + CIEI5(c_i) + CIEI6(c_i) + CIEI7(c_i) + CIEI8(c_i)$

$A\_CIS(c_i) = CIS1{:}CCM1(c_i) + CIS2(c_i) + CIS3(c_i) + CIS4(c_i) + CIS5(c_i) + CIS6(c_i) + CIS7{:}CCM2(c_i) + CIS8{:}CCM3(c_i)$

$A\_CDE(c_i) = CDE1{:}CCM1(c_i) + CDE2{:}CCM2(c_i) + CDE3{:}CCM3(c_i) + CDE4(c_i) + CDE5(c_i)$

Class External Relationships

$A\_CER(c_i) = CER1(c_i) + CER2(c_i) + CER3(c_i) + CER4(c_i) + CER5{:}CCM4(c_i) + CER6{:}CCM5(c_i) + CER7{:}CCM6(c_i)$

Class Connection Obscurity

$A\_CUR(c_i) = CUR1(c_i) + CUR2{:}CCM7(c_i)$

$A\_CDC(c_i) = CDC1{:}CCM8(c_i) + CDC2(c_i)$

$A\_CUER(c_i) = CUER1{:}CCM8(c_i) + CUER2{:}CCM4(c_i) + CUER3{:}CCM6(c_i) + CUER4{:}CCM5(c_i)$

$A\_CNI(c_i) = CNI1{:}CCM8(c_i) + CNI2(c_i) + CNI3(c_i)$

Class Dependency

$A\_CSI(c_i) = CSI1{:}CCM7(c_i) + CSI2(c_i)$

$A\_CEVR(c_i) = CEVR1(c_i) + CEVR2(c_i)$

$A\_CEFW(c_i) = CEFW1(c_i) + CEFW2(c_i) + CEFW3(c_i) + CEFW4(c_i)$

Object Interface Dependence

$A\_OIEI(c_i) = OIEI1(c_i) + OIEI2(c_i) + OIEI3(c_i) + OIEI4(c_i) + OIEI5(c_i) + OIEI6(c_i) + OIEI7(c_i) + OIEI8(c_i)$

$A\_OIS(c_i) = OIS1{:}OCM1(c_i) + OIS2(c_i) + OIS3(c_i) + OIS4(c_i) + OIS5(c_i) + OIS6(c_i) + OIS7{:}OCM2(c_i) + OIS8{:}OCM3(c_i)$

$A\_ODE(c_i) = ODE1{:}OCM1(c_i) + ODE2{:}OCM2(c_i) + ODE3{:}OCM3(c_i) + ODE4(c_i) + ODE5(c_i)$

Object External Relationships

$A\_OER(c_i) = OER1(c_i) + OER2(c_i) + OER3(c_i) + OER4(c_i) + OER5{:}OCM4(c_i) + OER6{:}OCM5(c_i) + OER7{:}OCM6(c_i)$

Object Connection Obscurity

$A\_OUR(c_i) = OUR1{:}OCM8(c_i) + OUR2{:}OCM9(c_i)$

$A\_ODC(c_i) = ODC1{:}OCM10(c_i) + ODC2(c_i)$

$A\_OUER(c_i) = OUER1{:}OCM8(c_i) + OUER2{:}OCM10(c_i) + OUER3{:}OCM5(c_i) + OUER4{:}OCM6(c_i)$

$A\_ONI(c_i) = ONI1{:}OCM10(c_i) + ONI2(c_i) + ONI3(c_i)$

Object Dependency

$A\_OSI(c_i) = OSI1{:}OCM9(c_i) + OSI2(c_i)$

$A\_OEVR(c_i) = OEVR1(c_i) + OEVR2(c_i)$

$A\_OEFW(c_i) = OEFW1(c_i) + OEFW2(c_i) + OEFW3(c_i) + OEFW4(c_i)$

4. Calculate the standard normal distribution of each aggregate calculated in the previous step, excluding the Zero class. For example,

   $ZA\_CIEI = (A\_CIEI - \mu) / \sigma$

   where μ is the mean of distribution A_CIEI and σ is its standard deviation.

5. Use the μ and σ calculated in the previous step to calculate the new values of the Zero class and object-class aggregates.

6. The following minor modularity sub-characteristics have associated sub-minor sub-characteristics.
   - Interface implementation dependence of classes (CIID) and objects (OIID)
   - Non-standard connection of classes (CNSC) and objects (ONSC)
   - State dependency of classes (CSD) and objects (OSD)

   For each of these minor modularity sub-characteristics, for each class and each object class, including Zero, calculate the sum of the normalised sub-minor aggregates calculated in the previous step.

   $A\_CIID(c_i) = ZA\_CIS(c_i) + ZA\_CDE(c_i)$

   $A\_CNSC(c_i) = ZA\_CUR(c_i) + ZA\_CNI(c_i)$

   $A\_CSD(c_i) = ZA\_CEVR(c_i) + ZA\_CEFW(c_i)$

   $A\_OIID(c_i) = ZA\_OIS(c_i) + ZA\_ODE(c_i)$

   $A\_ONSC(c_i) = ZA\_OUR(c_i) + ZA\_ONI(c_i)$

   $A\_OSD(c_i) = ZA\_OEVR(c_i) + ZA\_OEFW(c_i)$

7. Calculate the standard normal distribution of these sums, excluding Zero, to get an aggregate describing the minor modularity sub-characteristics.
   For example,
   $ZA\_CIID = (A\_CIID - \mu) / \sigma$
   where μ is the mean of distribution A_CIID and σ is its standard deviation.

8. Use the μ and σ calculated in the previous step to calculate the new values of the Zero class and object-class aggregates.

9. For each major modularity sub-characteristic, for each class and each object class, including Zero, calculate the sum of the associated minor modularity sub-characteristics.
   - Interface dependence of classes (CID) and objects (OID)
   - External relationships of classes (CMER) and objects (OMER)
   - Connection obscurity of classes (CCO) and objects (OCO)
   - Dependency of classes (CD) and objects (OD)

   $A\_CID(c_i) = ZA\_CIEI(c_i) + ZA\_CIID(c_i)$
   $A\_CMER(c_i) = ZA\_CER(c_i)$
   $A\_CCO(c_i) = ZA\_CUR(c_i) + ZA\_CDC(c_i) + ZA\_CNSC(c_i)$
   $A\_CD(c_i) = + ZA\_CSI(c_i) + ZA\_CSD(c_i)$

   $A\_OID(c_i) = ZA\_OIEI(c_i) + ZA\_OIID(c_i)$
   $A\_OMER(c_i) = ZA\_OER(c_i)$
   $A\_OCO(c_i) = ZA\_OVC(c_i) + ZA\_OUR(c_i) + ZA\_ODC(c_i) + ZA\_ONSC(c_i)$
   $A\_OD(c_i) = + ZA\_OSI(c_i) + ZA\_OIP(c_i) + ZA\_OSD(c_i)$

10. Calculate the standard normal distribution of these sums, excluding Zero, to get an aggregate describing the major modularity sub-characteristics.
    For example,
    $ZA\_CID = (A\_CID - \mu) / \sigma$
    where μ is the mean of distribution A_CID and σ is its standard deviation.

11. Use the μ and σ calculated in the previous step to calculate the new values of the Zero class and object-class aggregates.

12. For each class and each object class, including Zero, calculate the sum of the major modularity sub-characteristics.

$A\_CMOD(c_i) = ZA\_CID(c_i) + ZA\_CMER(c_i) + ZA\_CCO(c_i) + ZA\_COD(c_i)$
$A\_OMOD(c_i) = ZA\_OID(c_i) + ZA\_OMER(c_i) + ZA\_OCO(c_i) + ZA\_OOD(c_i)$

13. Calculate the standard normal distribution of these sums, excluding Zero, to get an aggregate describing modularity.

$ZA\_CMOD = (A\_CMOD - \mu) / \sigma$

where μ is the mean of distribution A_CMOD and σ is its standard deviation.

$ZA\_OMOD = (A\_OMOD - \mu) / \sigma$

where μ is the mean of distribution A_OMOD and σ is its standard deviation.

14. Use the μ and σ calculated in the previous step to calculate the new values of the Zero class and object-class modularity aggregate.

The result of this process is a set of normalised aggregate values describing the various sub-characteristics of modularity as well as a single aggregate value describing general modularity.

7.1.1.3 Class and object modularity weighted aggregate calculation

Fundamental to this calculation is the determination that each modularity sub-characteristic contributes equally to the aggregate value of its immediately associated sub-characteristic or characteristic. For example, the interface dependence, external relationships, connection obscurity and dependency sub-characteristics contribute equally to the levels of modularity present in the software. Weighting values are not included in the description of the aggregate calculation because they could occur in any place throughout the calculation. The particular purpose in taking the modularity measures should dictate the type of weighting used and should provide grounds to justify such weighting. Part of the process of content validation described in Section 1.4.1 of Chapter 1 involves assigning importance values to measured characteristics and features. These importance factors could be used to indicate where weighting could be applied during the aggregate calculation.

Using importance values to assign weighting in the aggregate calculation is one possibility.

Another possible weighting scheme could be based on the recognition that interface dependence modularity sub-characteristic describes an aspect of information hiding while the external relationships, connection obscurity and dependency all describe aspects of coupling. For information hiding to be equally represented with coupling in the aggregate calculation, the information hiding normalised aggregate could be weighted by a factor of 3 before being used to calculate the final modularity aggregate. Steps 12 to 14 of the aggregate calculation process would then become:

12. For each class and each object class, including Zero, calculate the sum of the major modularity sub-characteristics, with the interface dependence sub-characteristic weighted by a factor of 3.

    $A\_WEIGHTED\_CMOD(c_i) = 3(ZA\_CID(c_i)) + ZA\_CMER(c_i) + ZA\_CCO(c_i) + ZA\_COD(c_i)$
    $A\_WEIGHTED\_OMOD(c_i) = 3(ZA\_OID(c_i)) + ZA\_OMER(c_i) + ZA\_OCO(c_i) + ZA\_OOD(c_i)$

13. Calculate the standard normal distribution of these sums, excluding Zero, to get an aggregate describing modularity.

    $ZA\_WEIGHTED\_CMOD = (A\_WEIGHTED\_CMOD - \mu) / \sigma$
    where μ is the mean of distribution A_WEIGHTED_CMOD and σ is its standard deviation.
    $ZA\_WEIGHTED\_OMOD = (A\_WEIGHTED\_OMOD - \mu) / \sigma$
    where μ is the mean of distribution A_WEIGHTED_OMOD and σ is its standard deviation.

14. Use the μ and σ calculated in the previous step to calculate the new values of the Zero class and object-class modularity weighted aggregate.

Care should be taken when applying weighting values, that the values used are selected so that they provide the required amount of emphasis on the weighted measured sub-characteristics without overwhelming the contribution of the non-weighted measured sub-characteristics. A possible area for future research is the appropriate selection of weighting values in the calculation of the modularity aggregate.

7.1.2 Calculation of a Modularity Distance

Using the measured values of modularity to calculate the Euclidean distance between pairs of classes or objects provides an indication of the dissimilarity of modularity between the pairs. By taking advantage of the fact that the modularity measures are defined such that zero indicates optimum modularity and values above zero indicate decreasing modularity, an artificial Zero class and object-class can be added to the cases for which dissimilarity is calculated. The calculated levels of dissimilarity of all classes and object-classes with respect to this Zero class provide indications of relative modularity.

The first step in calculating the dissimilarity values is to perform the preliminary calculations previously described in Section 7.1.1.1. Next, all modularity measure distributions are normalised. Using a normalised distribution of measurement data ensures that each measure, regardless of its range, contributes equally to the distance value. As for the aggregate calculation previously described, the artificial Zero class and object-class should be normalised separately from the actual measures classes and object classes.

The standard normal distributions are calculated as follows (Swift 2001, p. 483).

$$Z\_X = (X - \mu_X) / \sigma_X$$

where μ_X is the mean of distribution X and σ_X is its standard deviation.

Using the previously calculated values of μ and σ, calculate the normalised value of the Zero class or object-class for measure X.

$$Z\_X_{Zero} = (x_{Zero} - \mu_X) / \sigma_X$$

Once all measure distributions are normalised, with a statistical software package such as SPSS, use all the modularity measure values to calculate the Euclidean distance between the Zero class or object-class and each of the measured system classes or object-classes. The greater the distance value, the less modular a class or object-class is.

The advantage of using the aggregate value to describe modularity is that its calculation takes into account the hierarchy of sub-characteristics that together describe modularity. The calculation process also results in intermediate aggregate values describing the various

modularity sub-characteristics. The calculation method also ensures that sub-characteristics described by more individual measures do not have a greater influence on the final modularity value than sub-characteristics described by only a few measures.

The Euclidean distance indicator of modularity has that advantage of being relatively easy to calculate as, once the measure distributions are normalised, a statistical software package such as SPSS will generate it automatically. Another advantage is that SPSS can use the Euclidean distances to perform analysis such as cluster analysis to identify classes and objects with similar levels of modularity, and can produce graphical representations, such as dendrograms, showing relative class and object-class modularity. A disadvantage of the Euclidean distance indicator of modularity is that its calculation, as described above, does not take into account the sub-characteristic hierarchy when calculating the final value. All measures are treated equally and sub-characteristics described by many measures will have a greater effect on the final indicator value than sub characteristics described by few measures.

In Section 7.3, an example of construct validation examines modularity aggregate values and Euclidean distance values as descriptors of general class and object modularity. This construct validation will investigate the validity of these two modularity indicators as descriptors of eMulePlus class modularity.

7.2. Content validation – eMulePlus software system

For the case studies described later in this chapter, content validation must demonstrate that the measures of C++ class and object modularity provide an adequate description of the modularity of the eMulePlus system. As Figure 5-4 of Chapter 5 shows, the first step in content validation is to examine the characteristics to be described by the measures and list the software features that affect the level of characteristics present. The **char**acteristic to **me**asure **r**elationship (CHARMER) diagrams listed in Table 6-4 of Chapter 6 support this step by displaying the natural language entity model point numbers associated with each identified modularity sub-characteristic. These natural language points describe features of the software that affect the levels of modularity sub-characteristic present in the software. The CHARMER diagrams associated with the content validation of the eMulePlus system are found in Appendix 4.

In the next step of content validation, the importance and frequency of occurrence of each of the characteristics and associated features are determined and stated. In this case study, all the

importance ratings are set to the same value because the aim is to describe general modularity, taking into account all the different modularity sub-characteristics described by the measures. Were some sub-characteristics more relevant to the study than others, then their importance ratings could be set higher than that of sub-characteristics of less importance. This would emphasise the need to demonstrate in the content validation that the defined measures adequately describe the more important sub-characteristics. The frequency ratings are set according to the frequency with which each identified software feature occurs within the software system. To determine the frequency ratings, the eMulePlus system source code is examined using the Understand for C++ code analysis application. The Appendix 4 eMulePlus CHARMER diagrams are annotated with the frequency ratings thus determined. Importance ratings are not specifically noted on these diagrams because, as stated previously, for this case study they are all set to the same value.

Once importance and frequency ratings have been assigned, the list of features identified as affecting the level of the characteristic present in the software is re-examined to make sure all crucial features are included, even if they occur infrequently. After examining the natural language models of C++ class and object modularity in Chapter 4, section 4.2, the features of the software identified as affecting the levels of C++ class and object modularity are judged to be sufficient for the purpose of this case study. Were the identified features judged insufficient, then the case study measurement of the eMulePlus system should not proceed.

The final step of content validation involves comparing each of the characteristics and features on the list to the individual measures that describe them to ensure that each important characteristic and frequently occurring feature is described by at least one measure. Examination of the C++ **class** modularity CHARMER diagrams shows that all but two identified software features are described by at least one measure implemented in the measurement instrument. These implemented measures are judged to adequately quantify their associated software features. Features 2.1.8 and 2.1.9 of class external relationships are not described by implemented measures. The existence of global functions within the scope of a class is described by feature 2.1.8. Examination of the eMulePlus system shows that it has a high frequency of global functions. Not measuring point 2.1.8 means that there is a high probability that the measured description of eMulePlus class external relationships will underestimate the level of external relationships for a class. The existence of global variables within the scope of a class is described by feature 2.1.9. Examination of the eMulePlus system shows that it has a low frequency of global variables. Not measuring point 2.1.9 means that

there is a low probability that the measured description of eMulePlus class external relationships will underestimate the level of external relationships for a class. Despite these omissions, the implemented measures of C++ class modularity will be declared to have sufficient content validity for the task of describing the modularity of the eMulePlus system.

Examination of the C++ **object** modularity CHARMER diagrams in Appendix 4 shows that determining the content validity of the implemented object modularity measures is not as straight forward a task as determining class content validity. As the Chapter 5, Figure 5-9 to 5-13 C++ object CHARMER diagrams indicate, in certain situations, some of the object modularity measures are unable to describe all the features of the software system. As was previously discussed in Chapter 5, measures CER2, OER5:OCM4, OER6:OCM5, OER7:OCM6 and OER8:OCM7 can be used to identify object-classes that are potentially not fully described by some measures of object modularity. Appendix 4 lists the CER2, OER5:OCM4, OER6:OCM5, OER7:OCM6 and OER8:OCM7 measured values taken from the eMulePlus system. The following sections discuss the content validation of the individual modularity sub-characteristics.

7.2.1 eMulePlus object interface dependence

As the Figure A4-5 CHARMER diagram of eMulePlus object interface dependence in Appendix 4 shows, when CER2 for an object-class is greater than zero, it is possible that the object interface dependence measures are unable to fully describe these object-classes. This situation arises when the object-class inherits more than once from the same ancestor class. For the eMulePlus system, measure CER2 indicates that 16 classes have distant ancestor classes that could cause this situation to arise. An examination, using the Understand for C++ code analyser, of the inheritance hierarchy of these 16 classes shows that in the eMulePlus system, no classes inherit from more than one version of an ancestor class. This means that the object-class E-R models are able to describe all the inherited elements of the object-classes in the eMulePlus system and the interface dependence measures with the CER2 validity indicator are able to fully describe the associated modularity sub-characteristic of the eMulePlus object-classes. The implemented measures of object interface dependence have sufficient content validity to provide an adequate description of the eMulePlus system.

7.2.2 eMulePlus object external relationships

As the Figure A4-6 CHARMER diagram of eMulePlus object external relationships in Appendix 4 shows, when CER2 for an object-class is greater than zero, it is possible that some of the object external relationships measures are unable to fully describe these object-classes. As discussed in the previous eMulePlus object interface dependence content validation in section 7.1.2.1, examination of the eMulePlus system shows that this situation does not arise. This means that measures OER1 and OER5:OCM4 are able to fully describe the eMulePlus system. Measure OER2 is not implemented and thus cannot measure the eMulePlus system. The feature quantified by measure OER2 occurs with moderate frequency in the software and so, not measuring it will not have a major impact on the final description of object external relationships. Since all other object external relationship measures can be made on the eMulePlus system, the implemented measures of object external relationships have sufficient content validity to provide an adequate description of the eMulePlus system.

7.2.3 eMulePlus object connection obscurity

As the Figure A4-7 and Figure A4-8 CHARMER diagrams of eMulePlus object connection obscurity in Appendix 4 show, in certain situations, some measures are unable to fully describe a software system. As previously discussed, measures annotated with the CER2 measure are able to fully describe the eMulePlus system and so, these implemented measures have sufficient content validity to adequately describe the variable connection, unstated relationships, distant connection and unexpected relationship sub-characteristics of eMulePlus object connection obscurity. The aspect of variable connection described by measure OVC2 cannot be described by the implemented measures. Since this feature occurs only with moderate frequency in the eMulePlus system, the decision is made to accept the remaining measure of variable connection as providing an adequate description of this sub-characteristic.

Figure 7-1 shows the CHARMER diagram of the non-standard connection sub-characteristic of object connection obscurity.

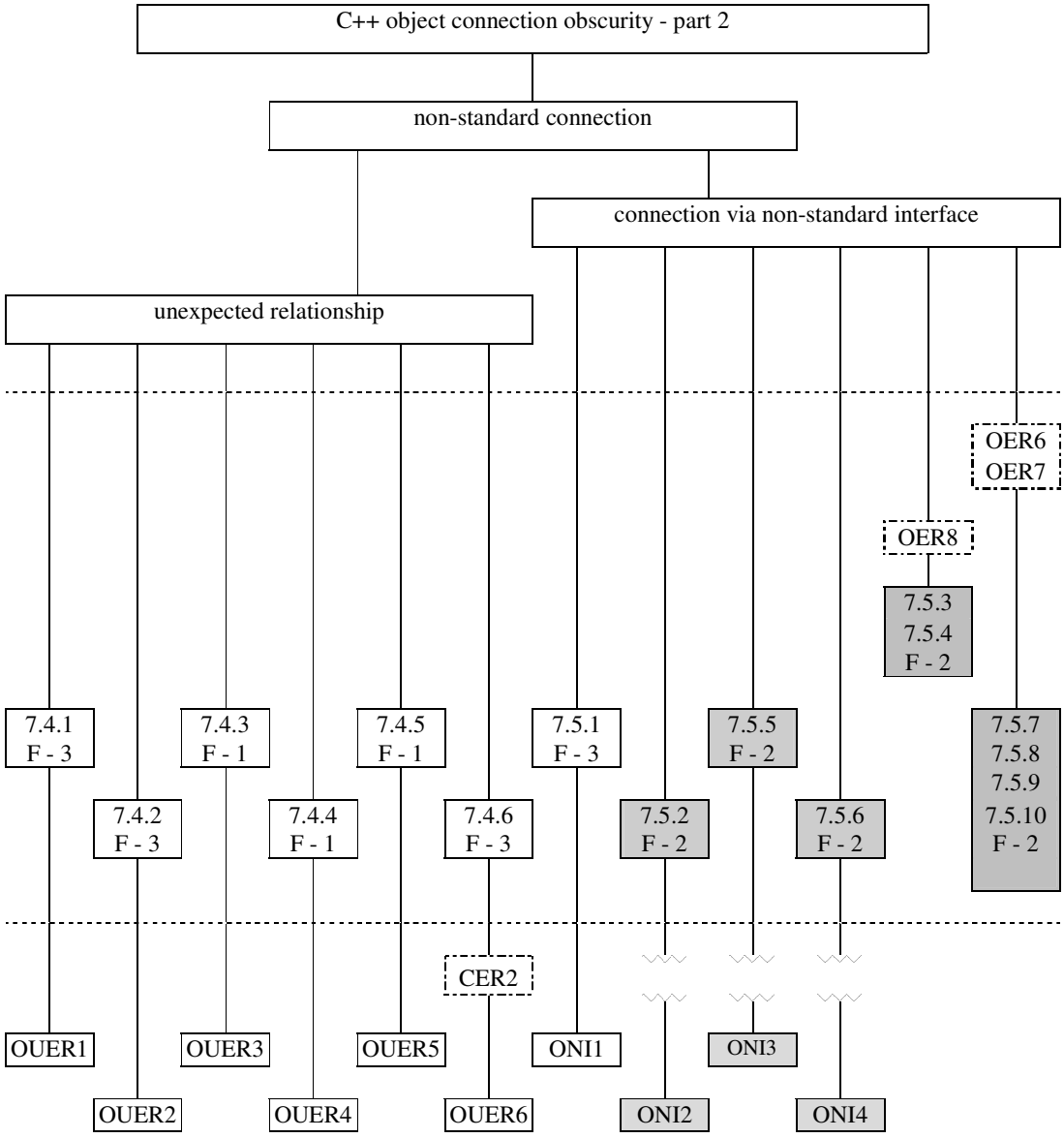


Figure 7-1 CHARMER diagram showing content validation for eMulePlus measures describing object connection obscurity.

From Figure 7-1 it can be seen that the implemented measures are able to fully describe eMulePlus object unexpected relationship sub-characteristic of modularity but are unable to fully describe the connection via non-standard interface sub-characteristic. The shaded boxes in the Figure 7-1 CHARMER diagram indicate software features that are not described by the measures implemented in the measurement instrument that will be applied to the eMulePlus software system. This diagram shows that only feature 7.5.1 of the ten identified features of the connection via non-standard interface sub-characteristic of object modularity is described by any implemented measure. Applying measures OER6:OCM5, OER7:OCM6 and OER8:OCM7 to the eMulePlus system shows that while no eMulePlus object-classes have friend global functions (OER7:OCM6), 29 object-classes have friend classes (OER6:OCM5) and 28 object-classes in the eMulePlus system are friends to other object-classes (OER8:OCM7). The classes identified by these measures as being involved in friend type relationships potentially have levels of connection via non-standard interface that reduce the modularity of the eMulePlus system. As the Figure 7-1 CHARMER diagram shows, no measures are defined to describe these relationships. This problem, combined with the inability of the selected measurement instrument to implement measures ONI2, ONI3 and ONI4, leads to the implemented measurement instrument being judged to have insufficient content validity to adequately describe the levels of connection via non-standard interface present in the eMulePlus system. This lack of content validity means that the connection via non-standard interface sub-characteristic of object connection obscurity should not be measured for the eMulePlus system.

7.2.4 eMulePlus object dependency

As the Figure 7-2 CHARMER diagram of eMulePlus object dependency shows, in certain situations, some measures are unable to fully describe a software system.

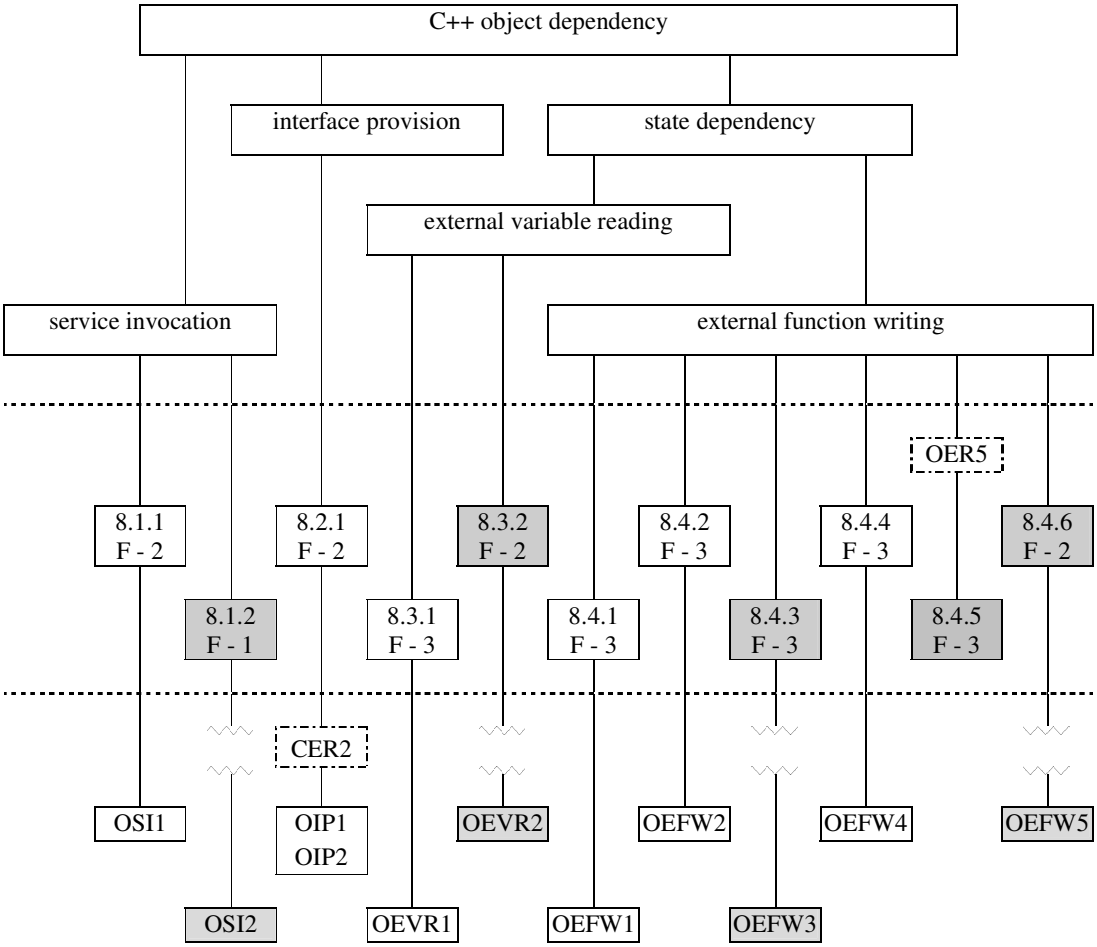


Figure 7-2 CHARMER diagram showing content validation for eMulePlus measures describing object dependency.

As previously discussed, measures annotated with the CER2 measure are able to fully describe the eMulePlus system and so, the implemented measures have sufficient content validity to adequately describe the interface provision sub-characteristic of eMulePlus object dependency.

When OER5:OCM4 for an object-class is greater than zero, the object-class has one or more static attributes. For the eMulePlus system, measure OER5:OCM4 indicates that only 1 object-class has a static attribute. Examination of this object-class, using the Understand for C++ code

analyser, shows that this static attribute is not written to by any object in the eMulePlus system. This means that the validity of the measured description of eMulePlus object dependency is not reduced by the lack of measures defined to describe feature 8.4.5 of object dependency since this feature does not occur in the eMulePlus system. The validity of the description of object modularity is however reduced by the inability of the measurement instrument to implement measures OSI2, OEVR2, OEFW3 and OEFW5. In particular, the Figure 7-2 CHARMER diagram shows that measure OSI2 describes a feature of the software that occurs with high frequency in the eMulePlus system. The measurement instrument is judged to have insufficient content validity to provide an adequate description of eMulePlus system object dependency. This lack of content validity means that object dependency should not be measured for the eMulePlus system.

Table 7-1 summarises the results of the content validation of the implemented measurement instrument with respect to the eMulePlus system.

Modularity sub-characteristic	Content validity	Proceed with measurement
class interface dependence	sufficient	yes
class external relationships	sufficient	yes
class connection obscurity	sufficient	yes
class dependency	sufficient	yes
object interface dependence	sufficient	yes
object external relationships	sufficient	yes
object connection obscurity: variable connection	sufficient	yes
object connection obscurity: unstated relationships	sufficient	yes
object connection obscurity: distant connection	sufficient	yes
object connection obscurity: non-standard connection: unexpected relationship	sufficient	yes
object connection obscurity: non-standard connection: connection via non-standard interface	insufficient	no
object dependency	insufficient	no

Table 7-1 Results of eMulePlus system content validation

The measurement instrument can be applied to the eMulePlus system to describe the aspects of modularity for which it is able to provide a sufficiently valid description. The implemented measurement instrument has sufficient content validity to provide an adequate description of

eMulePlus class modularity. It also has sufficient content validity to provide an adequate description of eMulePlus object interface dependence, external relationships and the variable connection, unstated relationship, distant connection and unexpected relationship sub-characteristics of object connection obscurity. The implemented measurement instrument has insufficient content validity to provide an adequate description of eMulePlus object dependency and the connection via non-standard interface sub-characteristic of object connection obscurity.

7.3. Example of a construct validation – eMulePlus software system

In general terms, construct validation involves measuring the characteristic of interest, in this case modularity, and characteristics that are theoretically associated with this characteristic. The strength of the relationship between the characteristic of interest and the associated characteristics is then statistically evaluated and from this, the construct validity of the measures of the characteristic of interest is estimated. The modularity measures defined in this thesis were developed based on Meyer's (1997, pp. 46-53) five rules of modularity. Associated with these five rules are five criteria of modularity. These are: decomposability, composability, understandability, continuity and protection (Meyer 1997, pp. 40-46). Were sufficiently valid and reliable measures of these five software criteria available, then they could be used to perform construct type validation of the modularity measures developed in this thesis. Since such modularity criteria measures are yet to be defined, this is an area for future work rather than a construct validation that can be demonstrated in this thesis.

An alternative to these five criteria is a construct validation of the modularity measures with respect to software size. Figure 1-6 in Section 1.4 described a process of construct validation. According to Steps 1 and 2 of this process, the theory associated with software modularity should be examined and software characteristics that would distinguish modules with differing levels of modularity be selected. One intuitive view of modularity is that it is related to module length. (Fenton 1995, p. 189) According to this view, longer (or larger) modules have lower levels of modularity. This view is supported by the object oriented design heuristic recommending that, to promote high levels of modularity, a software system be composed of several smaller modules rather than one single large module. The example of a construct validation described in this section will investigate the relationship between the modularity measures defined in this thesis and a measure of module size, in order to demonstrate evidence of construct validity of the modularity measures. This construct validation will be with respect

to class modularity only, as the previous section 7.2 identified the modularity measures as having sufficient content validity to describe class modularity but insufficient content validity to fully describe object modularity.

It is important to note that evidence of modularity measure construct validity obtained in this example is only applicable to the eMulePlus system with respect to a single size measure. This single piece of evidence is not sufficient to be able to declare the measures of class modularity to have an acceptable level of construct validity. To be able to declare the class modularity measures to have generally acceptable levels of construct validity, many diverse software systems will need to be evaluated against different measures of a number of criteria identified as being theoretically related to class and object modularity.

Another important point to note is that, should the construct validation procedure confirm a relationship between the class modularity measures and class size, this does not mean that the size measure can be substituted for the modularity measures as an indicator of class modularity. One reason for this is that while the modularity to size relationship may hold for a population of measured classes, it may not be true for each individual class in the population. A corollary of this is height vs weight example of construct validation discussed in section 1.4.2. While theory suggests that there is a relationship between height and weight of adults, and construct validation of a population of adults may support this theory, some individual adults within the sample population may not. For example, a very short, fat person may weigh the same as a much taller, thin person which means that a height measure cannot be substituted for a weight measure. Another reason why evidence of a relationship between modularity and size measures does not mean that size measure can be substituted for modularity measures is that, while size measures may indicate modules with high or low modularity, they do not specifically identify features of the module that contribute to overall modularity levels. A large size measure indicates that a module should be split into several smaller modules to increase its modularity however modularity measures will indicate other ways in which the module's modularity can be increased without necessarily dividing it into several modules.

Having identified the theoretical relationship between module size and modularity as the basis for an example construct validation, the next stages of the validation process can proceed. Steps 3 and 4 of the Figure 1-6 construct validation process require that a validation software system be selected and measures of module modularity and size be taken from this system. This sample construct validation will be performed on the eMulePlus software system. Modularity will be

represented as a Euclidean distance value calculated according to the process described in section 7.1.2 and as an unweighted aggregate value and a weighted aggregate value calculated according to the process described in section 7.1.1. The weighted aggregate is calculated in the manner described in section 7.1.1.3 where the class interface dependence values are multiplied by a factor of 3 to balance the three coupling aggregate values.

Figure 1.2 illustrates the measurement relationships between software source code size, lines of code and measures describing size by quantifying the lines of code software feature. In this construct validation example, a measure quantifying lines of code will be used to describe module size. The following equation defines this measure, based on the entity-relationship mathematical model of modularity developed in the entity modelling stage of measure development.

$$CLASS\_LOC = \{(ci, total)\} = \{(x.ci, total) \mid x \in C \wedge total = \sum\{y.lines \mid y \in M \wedge x.ci = y.ci\}\}$$

Examination of the defined modularity measures shows that only measure CIS5 quantifies the lines of code feature of C++ software. Since construct validation relies on correlation between measures of software characteristics, it would be advantageous to have the modularity and size characteristics described by separate features since using common features could artificially enhance the correlation between measures of the characteristics. Measure CIS5 provides such a link between the modularity and size measures. To demonstrate the possible effect measure CIS5 could have on the correlation, Figures 7-3, 7-4 and 7-5 show the relationship between unweighted and weighted class modularity aggregates and modularity Euclidean distances calculated with and without measure CIS5.

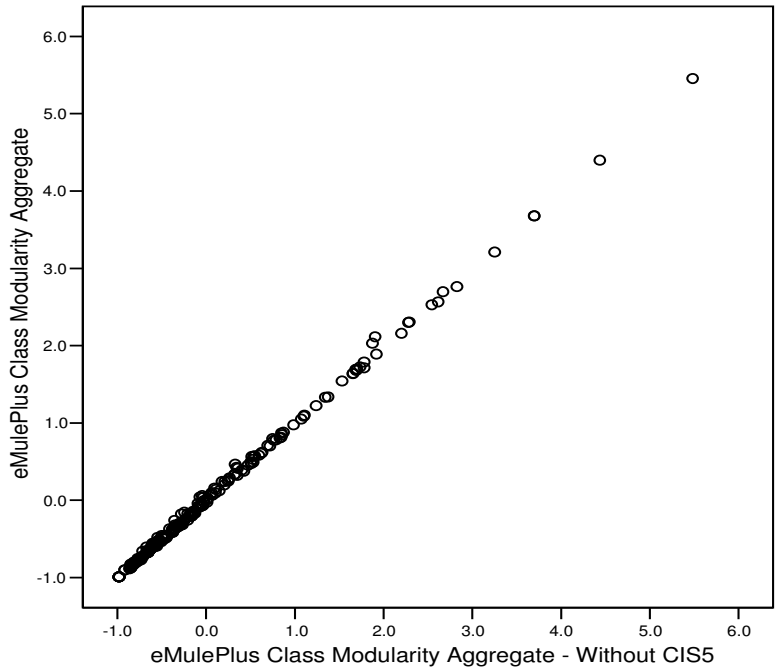


Figure 7-3 Effect of measure CIS5 on eMulePlus unweighted class modularity aggregate values

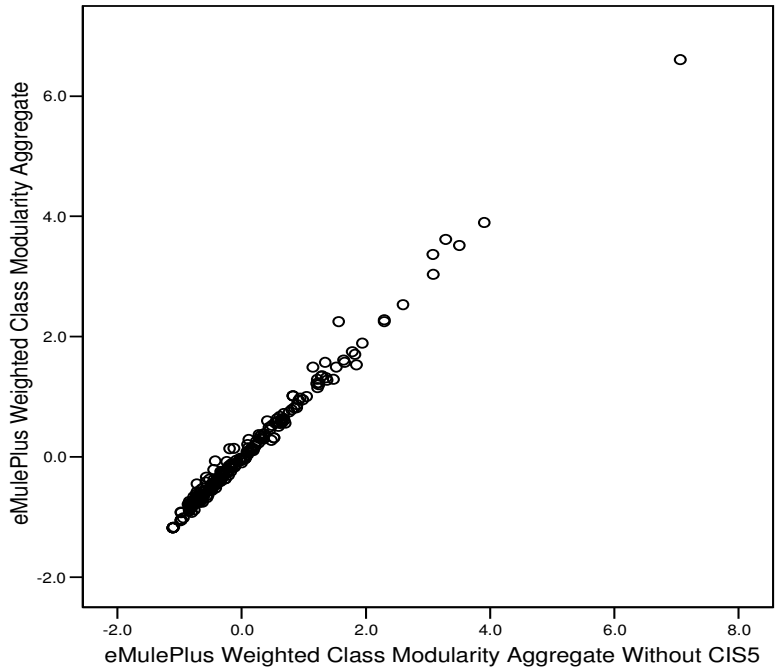


Figure 7-4 Effect of measure CIS5 on eMulePlus weighted class modularity aggregate values

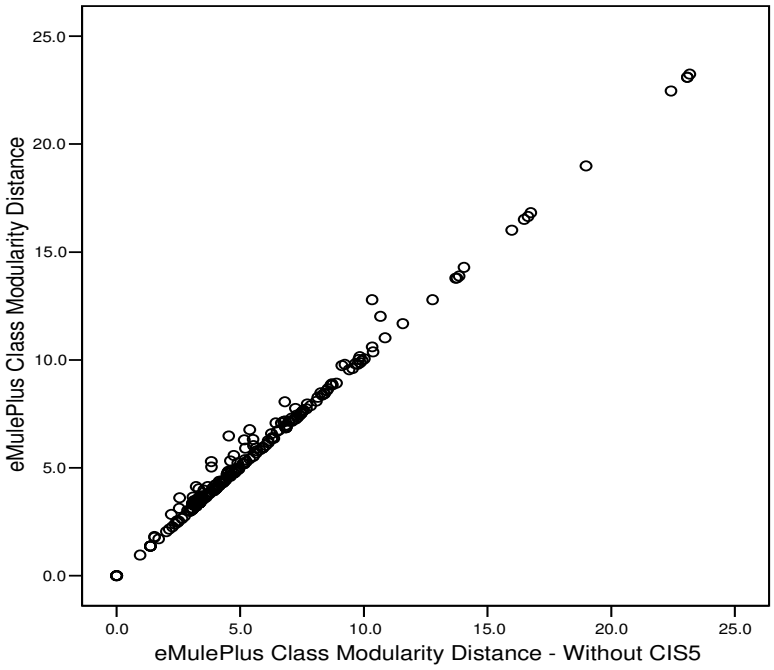


Figure 7-5 Effect of measure CIS5 on eMulePlus class modularity distance values

As these graph show, measure CIS5 has a relatively small effect on the final modularity values and CIS5's effect on the final construct validation correlation should also be small. For this reason, measure CIS5 will remain in the modularity aggregate and distance calculations for this construct validation. The values of lines of code, unweighted and weighted modularity aggregate and Euclidean distance for eMulePlus classes are listed in Appendix 5.

Although calculated by different methods, if both the unweighted and weighted modularity aggregates and modularity distance values provide a valid description of modularity, then there should be a strong relationship between their values for each measured class. Figure 7-6 illustrates the relationships between the weighted and unweighted modularity aggregates and modularity Euclidean distance values.

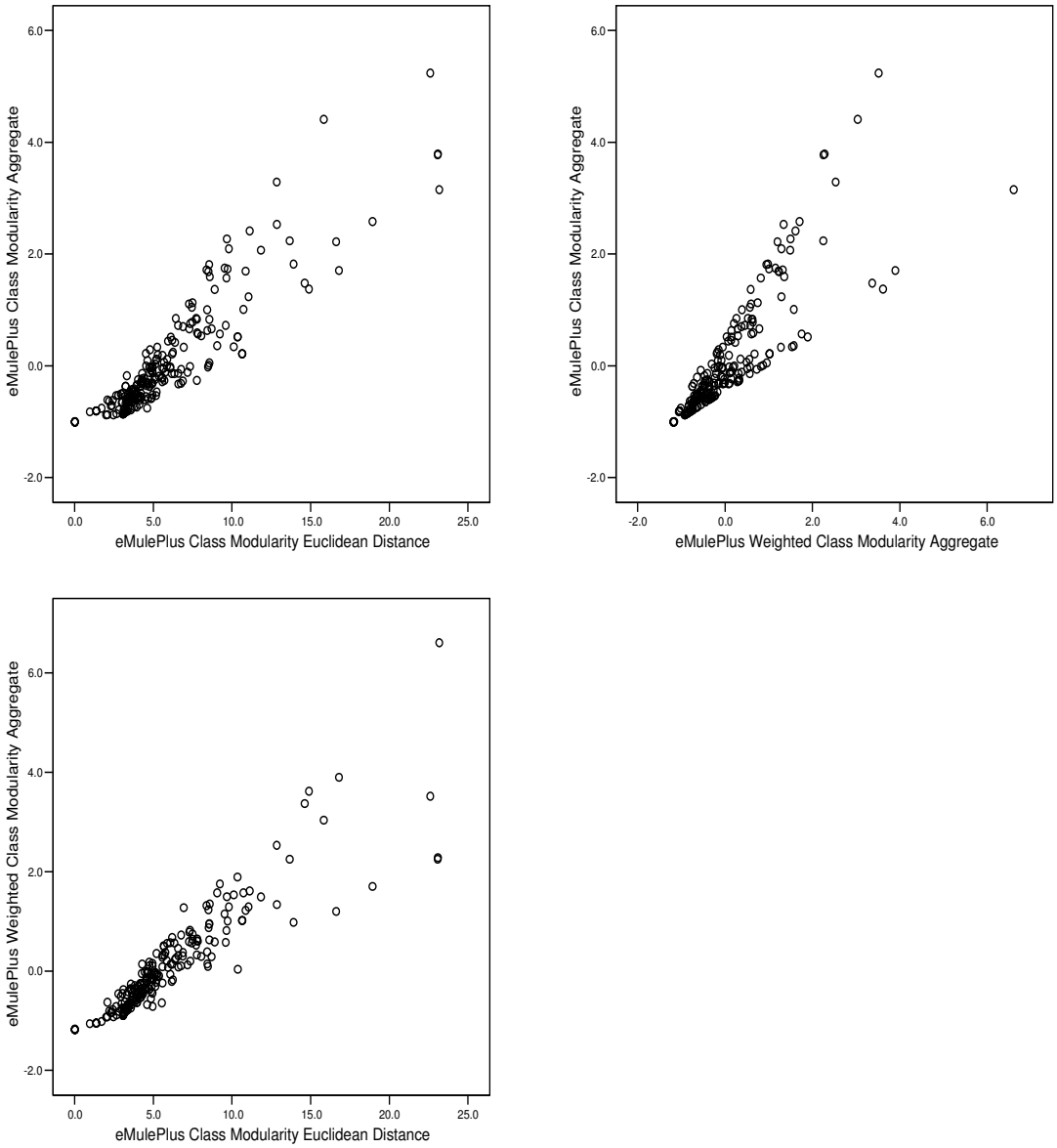


Figure 7-6 Comparison of weighted and unweighted modularity aggregate and modularity distance indicators of eMulePlus class modularity

Figure 7-6 shows a strong positive relationship between these three different modularity indicators which means it is highly likely that they are describing the same characteristic of the eMulePlus system software. The construct validation performed in the remainder of this section, will provide evidence as to whether or not this characteristic is modularity. As well as

providing evidence of the construct validity of the defined modularity measures, the construct validation described in the remainder of this section will also provide evidence as to which of these three methods of data reduction gives the most valid indication of eMulePlus software system class modularity.

According to Step 5 of the Figure 1-6 process of construct validation, the measurement data must be analysed using appropriate statistical techniques, to determine whether or not classes with high and those with low measured modularity are statistically differentiated by the class size measure. Table 7-2 describes the distributions of eMulePlus class total lines of code, modularity aggregate and modularity distance values. The artificial Zero class is not included in this description and will not be included in the construct validation because it does not represent a genuine case.

		eMulePlus Class Lines of Code	eMulePlus Class Modularity Aggregate	eMulePlus Weighted Class Modularity Aggregate	eMulePlus Class Modularity Euclidean Distance
N	Valid	249	249	249	249
	Missing	0	0	0	0
Mean		266.6988	.0000000	.0000000	5.6422
Median		136.0000	-.3142856	-.3064389	4.5300
Mode		.00	-.64350	-.71328	3.42
Std. Deviation		424.83571	1.00000000	1.00000000	3.80883
Variance		180485.381	1.000	1.000	14.507
Skewness		3.909	2.178	2.349	2.192
Std. Error of Skewness		.154	.154	.154	.154
Kurtosis		20.780	5.749	9.105	6.498
Std. Error of Kurtosis		.307	.307	.307	.307
Range		3543.00	6.24191	7.78536	23.18
Minimum		.00	-1.00601	-1.17927	.00
Maximum		3543.00	5.23590	6.60609	23.18
Percentiles	25	27.0000	-.6369538	-.6818372	3.4235
	50	136.0000	-.3142856	-.3064389	4.5300
	75	336.5000	.2158530	.3624061	6.8170
	95	962.5000	2.1564022	1.7276447	13.2605

Table 7-2 eMulePlus class lines of code, modularity aggregate and modularity distance distribution statistics

From Table 7-2 it can be seen that the class unweighted and weighted modularity aggregates,

modularity distance and lines of code distributions are positively skewed. This deviation from a normal distribution means that the construct validation must be performed with a nonparametric correlation technique. Comparison of the median and maximum values in Table 7-2 also shows that the unweighted and weighted modularity aggregates, distance and lines of code distributions contain outliers. Extreme outlier values should be removed from the data set prior to correlation analysis as they may have an undue effect on the final correlation coefficient obtained. Figure 7-7 shows eMulePlus outlier classes having total lines of code or unweighted modularity aggregate values above the 95th percentile, Figure 7-8 shows eMulePlus outlier classes having total lines of code or weighted modularity aggregate values above the 95th percentile and Figure 7-9 shows eMulePlus outlier classes with total lines of code or modularity distance values above the 95th percentile. All these classes, which can be identified from the Appendix 5 Table 5-1 listing of lines of code, unweighted and unweighted modularity aggregates and modularity distance values, will be excluded from the construct validation correlation calculations.

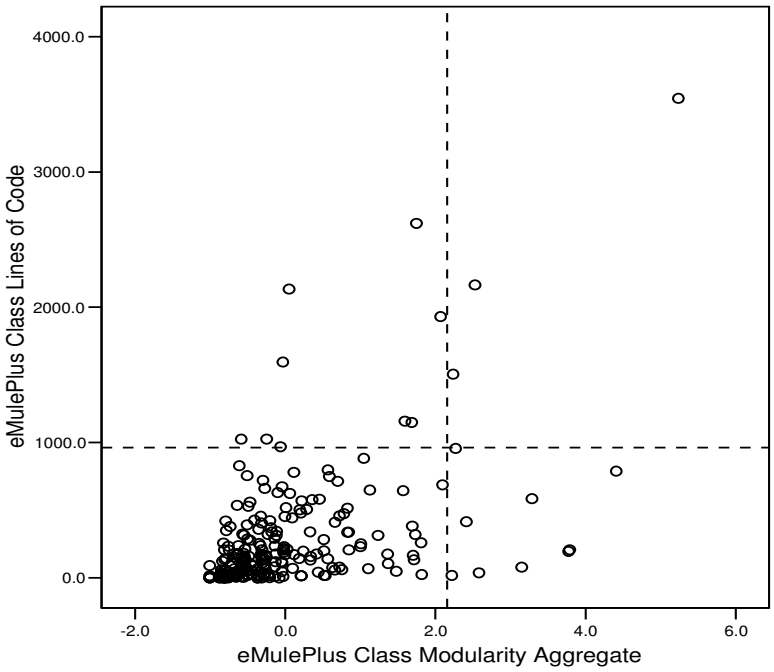


Figure 7-7 eMulePlus class lines of code and unweighted modularity aggregate outliers

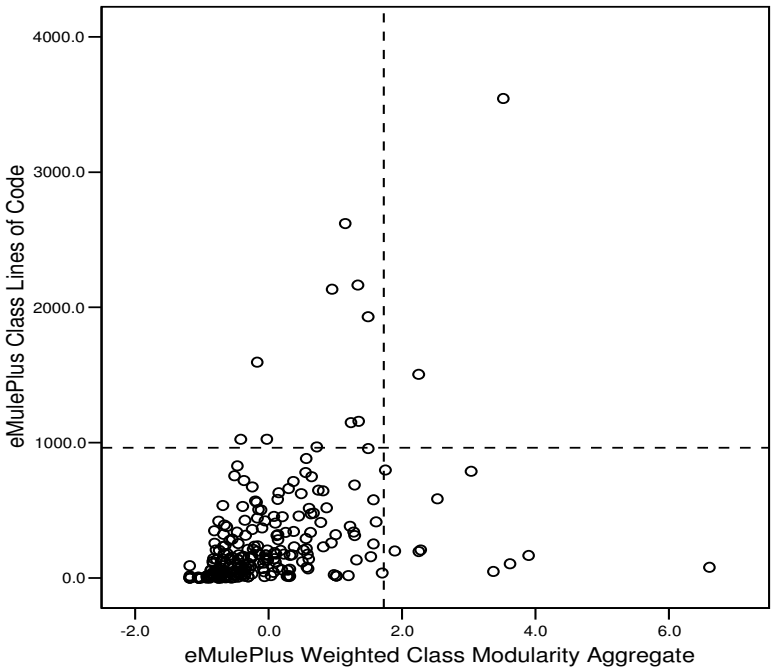


Figure 7-8 eMulePlus class lines of code and weighted modularity aggregate outliers

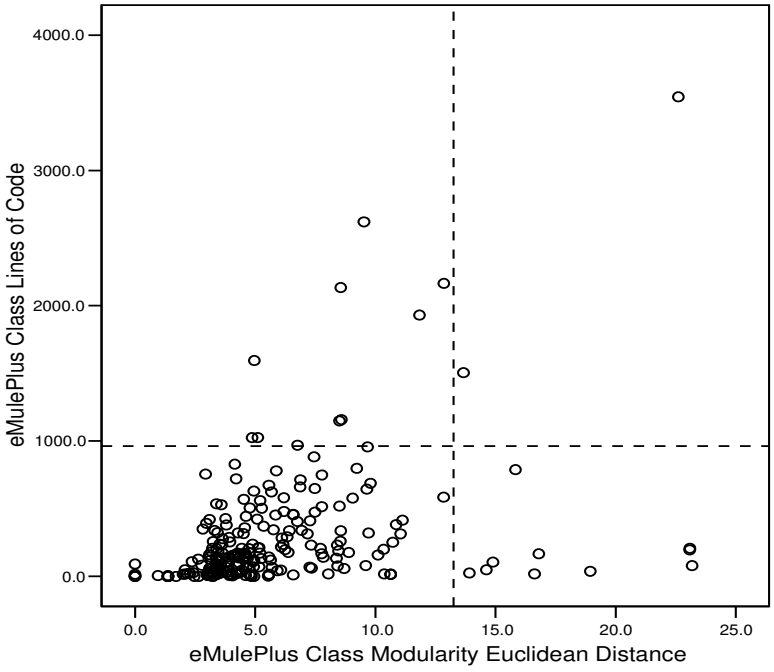


Figure 7-9 eMulePlus class lines of code and modularity Euclidean distance outliers

Table 7-3 describes the correlation relationship between the remaining eMulePlus class lines of code, unweighted and weighted modularity aggregates and modularity Euclidean distance values.

			eMulePlus Class Lines of Code	eMulePlus Class Modularity Aggregate	eMulePlus Weighted Class Modularity Aggregate	eMulePlus Class Modularity Euclidean Distance
Spearman's rho	eMulePlus Class Lines of Code	Correlation Coefficient	1.000	.515(**)	.556(**)	.474(**)
		Sig. (2-tailed)	.	.000	.000	.000
		N	222	222	222	222
	eMulePlus Class Modularity Aggregate	Correlation Coefficient	.515(**)	1.000	.917(**)	.903(**)
		Sig. (2-tailed)	.000	.	.000	.000
		N	222	222	222	222
	eMulePlus Weighted Class Modularity Aggregate	Correlation Coefficient	.556(**)	.917(**)	1.000	.934(**)
		Sig. (2-tailed)	.000	.000	.	.000
		N	222	222	222	222
	eMulePlus Class Modularity Euclidean Distance	Correlation Coefficient	.474(**)	.903(**)	.934(**)	1.000
		Sig. (2-tailed)	.000	.000	.000	.
		N	222	222	222	222

** Correlation is significant at the 0.01 level (2-tailed).

Table 7-3 Strength of relationships between eMulePlus class lines of code, unweighted and weighted modularity aggregates and modularity distance

Table 7-3 shows that there is a very strong positive relationship between the unweighted and weighted modularity aggregates and modularity Euclidean distance indicators of eMulePlus class modularity. This means that all these indicators are providing a similar summary of the modularity measured values and are therefore highly likely to be describing the same characteristic of the measured eMulePlus classes.

From Table 7-3 it can be seen that there is a moderately strong relationship between the unweighted and weighted modularity aggregate values and lines of code for the eMulePlus classes and between the modularity Euclidean distance value and lines of code. According to Step 6 of the construct validation process described in Figure 1-6, this is evidence that the unweighted and weighted aggregate values and the distance values calculated from the

modularity measures have a moderate degree of construct validity with respect to the theory that relates modularity and module size. It also offers evidence of the construct validity of lines of code as a measure of class size and of the validity of the theorised relationship between class size and modularity. The correlation between lines of code and the weighted modularity aggregate is slightly stronger than between lines of code and unweighted modularity aggregate which in turn is slightly stronger than the relationship between lines of code and modularity Euclidean distance. This is evidence that, for the eMulePlus software system, the weighted modularity aggregation provides a more valid summary of the modularity measures than the unweighted aggregate and Euclidean distance calculations. Further construct validations against different software systems and criterion characteristics are needed before this can be conclusively determined.

Given that the aggregate and distance values calculated from the eMulePlus class modularity measures are positively related to the lines of code size measure, the question arises as to whether the individual measures from which they are calculated are also positively related to the lines of code measure. Appendix 3 Table 3-1 shows the correlation between a class total lines of code count and measured values of class modularity taken from eMulePlus classes. As expected, measure CIS5 is strongly related to the lines of code measure. This is expected because CIS5 is itself a lines of code count.

From Appendix 3 Table 3-1, measures that are strongly related to the lines of code measure are

- CIS5 - median lines of code in class interface methods
- CUR2:CCM7 and CSI1:CCM7 - number of global functions invoked by class methods
- CSI2 - number of other classes whose methods are invoked by a class
- CDE4 - number of methods writing to a same class attribute
- CDE5 - number of methods reading from a same class attribute

The remaining measures of class modularity are not strongly related to the lines of code measure and so, are providing a description of modularity that is independent of the number of lines of code in the class.

While the individual measures of class modularity are not uniformly related to the class size lines of code measure, the composite modularity indicators of unweighted and unweighted modularity aggregates and modularity Euclidean distance are related to size. This is in

accordance with software engineering theory, as expressed by Fenton (1995, p. 189), used as the basis for the preceding construct validation of the class modularity measures. This means that while the modularity composite measures are related to module size, the individual measures from which they are calculated offer more insight into the modularity of individual classes than may be obtained from a lines of code measure describing class size. This result is true for modularity and size measures describing the eMulePlus software system. Before this result can be declared to be generally true, it needs to be evaluated for several different software systems.

For the purposes of the following case studies, the previous content and construct validations will be accepted as providing sufficient evidence of the validity of the defined measures of C++ class and object modularity. These two case studies describe two types of measurement and data analysis of eMulePlus system modularity. The first case study uses the unweighted modularity aggregate values to identify eMulePlus classes and objects with low modularity. The unweighted modularity aggregate summary of the measures is selected instead of the weighted modularity aggregate or Euclidean distance because it represents most closely the hierarchical structure of the modularity sub-characteristics, described in Chapter 3, which formed the basis of the measure development. The second case study presents a detailed description of the interface dependence of a selected class identified in the previous case study as having low modularity.

7.4. Modularity measurement case studies

Processes such as the GQM paradigm (Basili 1988) place software measurement within a wider context of the purpose for which the measures are being applied. As illustrated in Figure 2-2, the work in this thesis assumes that this contextualisation has already been performed and the characteristics to be described by the measures identified. In this case study, the simple aim in making the measures is to identify class and object modules with relatively low modularity. In practice, this simple aim would be part of a higher level aim such as to improve software maintainability or reusability. This higher level aim involves establishing the validity of the relationship between modularity and another characteristic of interest such as maintainability to show that the modularity measures can predict software maintainability. Such a case study, while very relevant to the real world of software engineering, is beyond the scope of this thesis. This thesis aims to identify a way in which descriptive measures of low level characteristic such as modularity can be developed such that they provide a valid description of the software. For this reason, the case studies presented in this chapter will be presented in the limited context of describing aspects of C++ class and object modularity. In future, should the process of measure development advocated in this thesis be accepted in the software community, further research could be performed to place the systematic measure development process within the wider context of software quality assessment and control.

The first step of the measure application process is to state the aim of the measurement activity; what the user wishes to find out about the software system from the measurement data collected. The next step is to examine the validity of the measures and decide whether or not the measures implemented within the measurement instrument provide an adequate description of the software system characteristic of interest. The adequacy of the description is determined with respect to the previously stated aims of the measurement study and the composition of the software system to be measured. If the measures are determined to have sufficient validity, the next step is to apply the measurement instrument to the software system source code and obtain the measurement data. This data is then analysed and the resulting information presented and interpreted with respect to the original aims of the measurement study.

The first case study presents a general description of the eMulePlus system class and object modularity. From this general view of the software system, individual class and object modules with relatively low modularity are identified. The second case study presents a more detailed

description of the interface dependence of a single class and object module. A graphical representation of the interface dependence of the module provides an overview that is further clarified by a detailed analysis of individual measures describing module interface size. This analysis serves to identify specific ways in which the implementation of the module can be modified to decrease its levels of interface dependence and hence improve its modularity.

These case studies demonstrate different types of analysis. They show how the highly detailed description of C++ class and object modularity obtained directly from the measures can be analysed and presented to provide different descriptions of the eMulePlus system from the general overview presented in case study 1 to the most specific description presented in case study 2.

7.4.1 Case study 1 - modularity of the eMulePlus software system

In this case study, a general description of the levels of class and object modularity in the eMulePlus software system is presented and discussed.

7.4.1.1 Aims

The aim of this case study is to obtain a general description of the modularity of the eMulePlus software system, highlighting any classes and objects with relatively low modularity. These could be candidates for further investigation to identify ways in which their modularity could be improved.

7.4.1.2 Application

The measurement instrument described in section 6.2 of Chapter 6 was applied to the eMulePlus system and the measured data was successfully extracted. Data was successfully imported from the eMulePlus source code into the C++ basic software model database. This basic software model was successfully transformed into the software modularity measurement database. While most transformations were accomplished smoothly and quickly, transformations 8, 9 and 10, defined in Appendix 2, that generate sets MIOCReadA, MIOCWriteA and MIOCInvM took several hours to complete when run on an IBM Pentium 3 laptop. The greater the number of classes and methods in a software system, the longer these transformations take to run. Measuring the eMulePlus system tested the limits of the implemented measurement instrument. To measure a larger system, a better measurement

instrument will be needed. Alternatively, the different sub-systems of which large software systems are most usually comprised, could be treated as software systems in their own right, and measured individually.

The modularity measures were taken from the eMulePlus software measurement database and the resulting data imported into the Statistical Package for the Social Sciences (SPSS) application for analysis.

7.4.1.3 Analysis and interpretation

The class and object modularity analyses present, for illustrative purposes, two different levels of analysis detail. The eMulePlus class modularity analysis is performed at a general level resulting in a single aggregate value of modularity. The eMulePlus object modularity analysis presents aggregated values of object interface dependence, external relationships and connection obscurity as well as a partial aggregate of object modularity calculated from these sub-characteristic aggregates. This extra information identifies object-classes with low modularity due to low levels of the measured sub-characteristics. Modularity aggregates are calculated according to the method defined in section 7.1. No weighting is used in this calculation. The eMulePlus modularity aggregate values used in this case study are listed in Appendix 5.

7.4.1.3.1 eMulePlus system class modularity

Table 7-4 describes the distributions of the modularity aggregates for the eMulePlus system classes. None of the distributions are normal and all have a strong positive skew. Comparison of the 75th percentile and maximum values indicates that extreme outlier classes exist that whose aggregate values indicate low modularity.

		eMulePlus Class Modularity Aggregate	eMulePlus Class Interface Dependence Aggregate	eMulePlus Class External Relationships Aggregate	eMulePlus Class Connection Obscurity Aggregate	eMulePlus Class Dependency Aggregate
N	Valid	249	249	249	249	249
	Missing	0	0	0	0	0
Mean		.0000000	.0000000	.0000000	.0000000	.0000000
Median		-.3142856	-.2770341	-.1743744	-.4159969	-.2895275
Mode		-.64350	-.92170	-.64663	-.41600	-.49616
Std. Deviation		1.00000000	1.00000000	1.00000000	1.00000000	1.00000000
Variance		1.000	1.000	1.000	1.000	1.000
Skewness		2.178	3.988	2.377	5.391	4.208
Std. Error of Skewness		.154	.154	.154	.154	.154
Kurtosis		5.749	23.629	7.180	37.134	22.673
Std. Error of Kurtosis		.307	.307	.307	.307	.307
Range		6.24191	9.14861	6.48465	8.58055	7.60244
Minimum		-1.00601	-.92170	-.64663	-.41600	-.49616
Maximum		5.23590	8.22691	5.83802	8.16455	7.10628
Percentiles	25	-.6369538	-.5143310	-.6466280	-.4159969	-.4961632
	50	-.3142856	-.2770341	-.1743744	-.4159969	-.2895275
	75	.2158530	.1850939	.2890585	-.0268967	.0079100
	95	2.1564022	1.5989198	2.2297711	1.5203834	1.7744980

Table 7-4 eMulePlus system class modularity aggregate statistics

The Figure 7-10 histogram describes the distribution of the eMulePlus system class modularity normalised aggregate values. This distribution has a mean of zero and a standard deviation of one. Low aggregate values indicate high modularity and increasing aggregate values indicate decreasing modularity. The reference lines at the 75^{th} and 90^{th} percentiles show that the majority of eMulePlus classes have relatively high modularity with only a few classes having a modularity aggregate indicating low modularity.

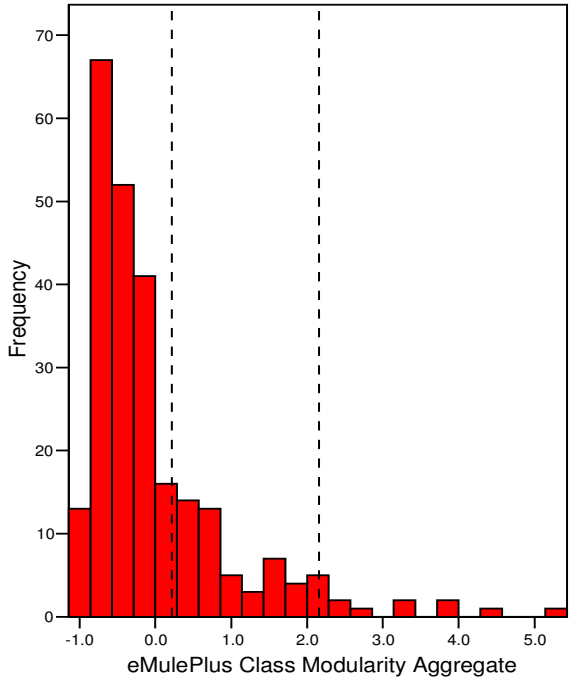


Figure 7-10 Frequency histogram of eMulePlus class modularity normalised aggregate values

Examination of the eMulePlus class modularity normalised aggregate values listed in Appendix 5 shows that 12 out of the total of 249 measured eMulePlus classes have an aggregate class modularity value above the 95th percentile. Class CPartFile with a class modularity aggregate of 5.24 has the lowest overall modularity of all the eMulePlus classes. Table 7-5 lists the outlier eMulePlus classes with a modularity aggregate value above the 95th percentile.

eMulePlus class	Class modularity aggregate
CPartFile	5.24
CKnownFile	4.41
COptionTreeFontSelColorButton	3.79
CColorButton	3.77
CClientReqSocket	3.29
CFriend	3.15
CAbstractFile	2.58
CWebServer	2.52
CEMSocket	2.41
CSearchDlg	2.27
CSharedFilesCtrl	2.24
DbEnv	2.22

Table 7-5 eMulePlus modularity aggregates of classes with relatively low modularity

The classes listed in Table 7-5 are all candidates for further investigation. The case study described in section 7.2.4.1 is an example of the type of investigation that could be performed on these classes to identify ways in which their levels of modularity could be improved.

7.4.1.3.2 eMulePlus system object modularity

Table 7-6 describes the distributions of the modularity aggregates for the eMulePlus system object-classes. As for the eMulePlus classes, none of the distributions are normal and all have a strong positive skew. Comparison of the 75th percentile and maximum values indicates that extreme outlier object-classes exist that whose aggregate values indicate low modularity.

		eMulePlus Partial Object Modularity Aggregate	eMulePlus Object Interface Dependence Aggregate	eMulePlus Object External Relationships Aggregate	eMulePlus Object Connection Obscurity Aggregate
N	Valid	249	249	249	249
	Missing	0	0	0	0
Mean		.0000000	.0000000	.0000000	.0000000
Median		-.3496998	-.2899236	-.3490133	-.3390886
Mode		-.81843	-.65407	-.49163	-.53396
Std. Deviation		1.00000000	1.00000000	1.00000000	1.00000000
Variance		1.000	1.000	1.000	1.000
Skewness		3.192	5.471	4.174	4.387
Std. Error of Skewness		.154	.154	.154	.154
Kurtosis		15.826	42.093	22.935	26.361
Std. Error of Kurtosis		.307	.307	.307	.307
Range		8.44795	9.32633	8.56114	8.04635
Minimum		-.81843	-.65407	-.49163	-.53396
Maximum		7.62951	8.67226	8.06950	7.51239
Percentiles	25	-.6003406	-.4519868	-.4916339	-.5339600
	50	-.3496998	-.2899236	-.3490133	-.3390886
	75	.2685776	.2022586	-.0041828	.1001530
	95	2.0680979	1.1695865	1.8790978	1.6656312

Table 7-6 eMulePlus system object modularity aggregate statistics

The histograms in Figure 7-11 describe the distribution of the eMulePlus object interface dependence, external relationships and connection obscurity modularity aggregates. The reference lines at the 75th and 90th percentiles show that the majority of eMulePlus object-classes have relatively high modularity with only a few object-classes having a modularity aggregate indicating low modularity. As discussed in section 7.1.2, the implemented measures have insufficient validity to describe the connection via non-standard interface sub-characteristic of connection obscurity and the dependency modularity sub-characteristic. Connection via non-standard interface is a minor sub-characteristic of object connection obscurity and so, the decision has been made to calculate a connection obscurity aggregate

without its input, which is relatively small. Dependency however is an immediate sub-characteristic of modularity and makes a significant contribution to the modularity aggregate values. Without measures describing object dependency, a full object modularity aggregate cannot be calculated.

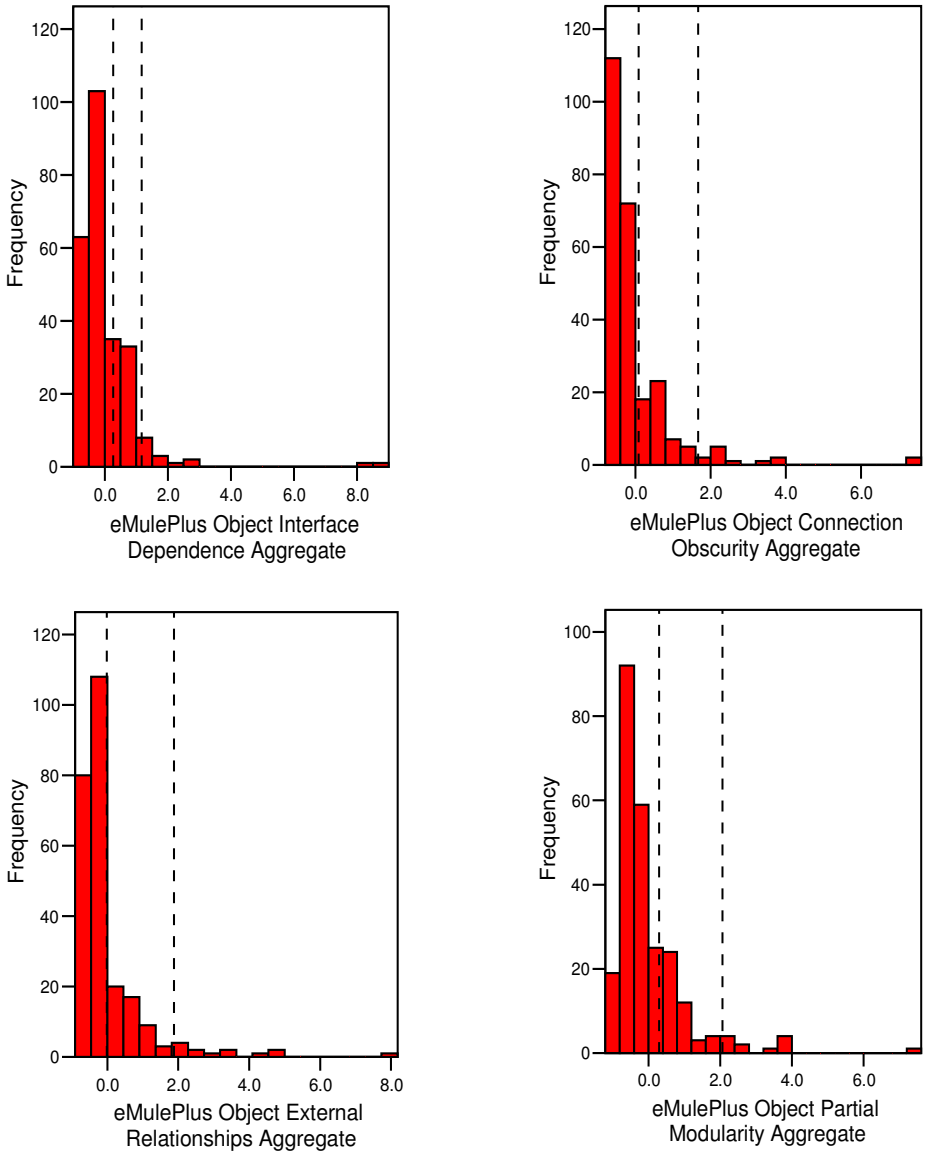


Figure 7-11 Frequency histogram of eMulePlus object modularity aggregates

The histograms of Figure 7-11 include a partial normalised object modularity aggregate histogram. This identifies eMulePlus object-classes that have the potential to have low object modularity. It is important to recognise that without a dependency component to this aggregate, an object-class with a high aggregate value cannot be said to have low modularity and an object-class with a low aggregate value cannot be said to have high modularity. In practice, the partial object modularity aggregate could be used to identify object-classes with the potential to have low modularity. These object-classes could be examined by hand to determine their levels of dependency and thus their levels of object modularity. Examination of the eMulePlus partial object normalised modularity aggregate listed in Appendix 5 shows that 12 eMulePlus object-classes have an aggregate value above the 95th percentile. Object-class CFriend with a partial object-class modularity aggregate of 7.63 is indicated as potentially having the lowest overall modularity of all the eMulePlus object-classes. Table 7-7 lists the object interface dependence, external relationships and connection obscurity aggregate values for the eMulePlus object-classes with a partial object modularity aggregate value above the 95th percentile. The case study described in section 7.2.4.2 is an example of the type of investigation that could be performed on these object-classes to identify ways in which their levels of modularity could be improved.

eMulePlus Object-Class	eMulePlus Partial Object Modularity Aggregate	eMulePlus Object Interface Dependence Aggregate	eMulePlus Object External Relationships Aggregate	eMulePlus Object Connection Obscurity Aggregate
CFriend	7.63	8.67	4.92	2.06
CColorButton	3.74	-.43	.60	7.51
COptionTreeFontSelColorButton	3.73	-.45	.60	7.51
COScopeCtrl	3.65	8.40	-.36	-.53
CPreferences	3.64	-.36	8.07	-.24
CUpDownClient	3.53	.42	4.81	2.02
DbEnv	2.72	-.32	4.43	1.47
CClientReqSocket	2.61	-.35	1.90	3.80
CemuleApp	2.25	.76	.02	3.83
CAsyncSocketEx	2.22	-.03	3.22	1.35
CServerSocket	2.15	-.35	1.45	3.32
CWebServer	2.15	-.46	3.54	1.33

Table 7-7 eMulePlus normalised modularity aggregates of object-classes with potentially low modularity

7.4.1.4 Discussion

This case study highlights the importance of performing a content validation of the measures with respect to the software system to be measured. Without a clear demonstration of measure content validity, it is difficult to determine whether or not a set of measures provide an adequate description of a particular software system. Demonstrating sufficient content validity means that measures can be confidently applied to a particular measurement situation and the resulting data analysed and interpreted. Measures determined to have insufficient content validity should not be applied to the task of describing a software system. Their lack of content validity should be documented and taken into account when analysing and interpreting the data from measures that were collected. This was the case when applying the measures of object modularity to the eMulePlus system. These measures were determined to have insufficient content validity to adequately describe the dependency sub-characteristic of object modularity.

Analysis and interpretation of the measurement data that was obtained to describe eMulePlus object modularity needs to take into account the fact that this aspect of modularity was not described.

Fundamental to the content validation is the clear connections established between the characteristics to be described by the set of measures, the features of the software believed to affect the levels of these characteristics present in the software and the measures defined to quantify these features. The descriptive measure development process described and demonstrated in this thesis supports measure content validation by documenting these connections throughout the development process. For example, it is possible to see from the Figure 7-1 CHARMER diagram of object connection obscurity, that the features identified in points 7.5.3, 7.5.4, 7.5.7, 7.5.8, 7.5.9 and 7.5.10 of the object modularity natural language model are not quantified by measures. This in turn shows that the aspects of connection via non-standard interface affected by these features are not described by measures.

When measuring a complex, cluster type characteristic (Gasking 1960; Ellis 1966) such as modularity (Meyer 1997, p39), a large volume of data may be produced due to the large number of measures defined to quantify the many features identified as affecting the many sub-characteristics identified. As previously noted, measuring the modularity of the eMulePlus system resulted in more than 25,000 points of data. It is important to select an analysis technique that reduces this large amount of data to useful information that can be interpreted. The large numbers of measures developed in this thesis provide a detailed description of C++ class and object modularity. This volume of data can be reduced to manageable proportions by the process of data analysis. The result of this is a more general, high level description of modularity. If a more detailed description is required, then there is sufficient information available, within the raw data set, to generate this too. It is possible to use analysis to reduce the detail of measured description but it is not possible to add detail by analysis. For this reason, it is useful to define measures to provide a detailed description of a characteristic rather than a more general one. As this case study has shown, it is possible to obtain a general description of eMulePlus system class and object modularity from the detailed measured description by aggregating the normalised measured values of the modularity sub-characteristics. Once a general description of modularity has been used to identify classes and object with relatively low modularity, a more detailed description of their modularity could be used to identify ways in which their modularity could be improved. The next case study demonstrates the detailed

description of software that can be obtained from the measures of C++ class and object modularity developed in this thesis. The natural language entity model from which the measures were developed supports the interpretation of this detailed description.

7.4.2 Case study 2 - interface dependence of CPartFile class and object modules

The results of the content validation summarised in Table 7-1 indicate that the implemented measures of class and object interface dependence provide an adequate description of eMulePlus system class and objects. Thus the measures of C++ class and object interface dependence provide an adequate description of the CPartFile class and object modules.

In this case study, interface dependence data of the eMulePlus system CPartFile class and CPartFile object will be analysed and interpreted to answer the question "How can the interface dependence be changed to increase the modularity of the CPartFile class and CPartFile object?". This case study demonstrates the analysis and interpretation of interface dependence measures to provide a detailed description of an aspect of the eMulePlus system. This complements the previous case study that demonstrated an analysis that provided a general description of eMulePlus system modularity. The CPartFile module has been selected for this detailed description case study because it was highlighted in the general description case study as having relatively low class modularity. The detailed description of CPartFile interface dependence that is obtained in this case study can be used to identify ways in which the modularity of the CPartFile class and object modules can be improved.

7.4.2.1 Aims

This case study aims to investigate the interface dependence of the eMulePlus system CPartFile class and object-class modules and identify specific ways in which the interface dependence of these modules can be decreased to improve their modularity.

7.4.2.2 Application

CPartFile class interface dependence is described by 1935 individual points of data and CPartFile object interface dependence is described by 2605 individual points of data. This measurement data is listed in Appendix 6. In its raw state, this large amount of data cannot be fully comprehended. It needs to be analysed so that useful information regarding the level of CPartFile interface dependence is obtained. This information can then be interpreted to gain an

understanding of CPartFile interface dependence.

7.4.2.3 Analysis and interpretation

The method of analysis used is this case study compares CPartFile interface dependence with that of an ideal module with low interface dependence. The derivation of this ideal module representation is discussed in section 3.2.2.2 of Chapter 3.

Figure 7-12 is a diagrammatic representation of the low interface dependence module described by the C++ class interface dependence natural language entity model in Chapter 4, section 4.2.3.2.1. This is the same representation presented in Figure 3-3. In accordance with point 1.2 of the C++ class interface dependence natural language entity model in Chapter 4, section 4.2.3.2.1, the Figure 7-12 module interface is small; containing no attributes and relatively few member methods. The interface methods themselves have relatively few lines of code, few same module method invocations and few same module attribute accesses. In accordance with point 1.1 of the C++ class interface dependence natural language entity model in Chapter 4, section 4.2.3.2.1, Figure 7-12 module interface methods do not directly or indirectly access same module interface attributes and do not directly or indirectly invoke other same module interface methods. This is indicated by the separation between interface methods shown in Figure 7-12 and the arrows from interface to hidden methods indicating that interface methods can invoke hidden methods but that hidden methods cannot invoke interface methods. In accordance with point 1.3 of the C++ class interface dependence natural language entity model in Chapter 4, section 4.2.3.2.1, there are no attributes in the Figure 7-12 module interface and the hidden attributes are not directly accessed by the interface methods.

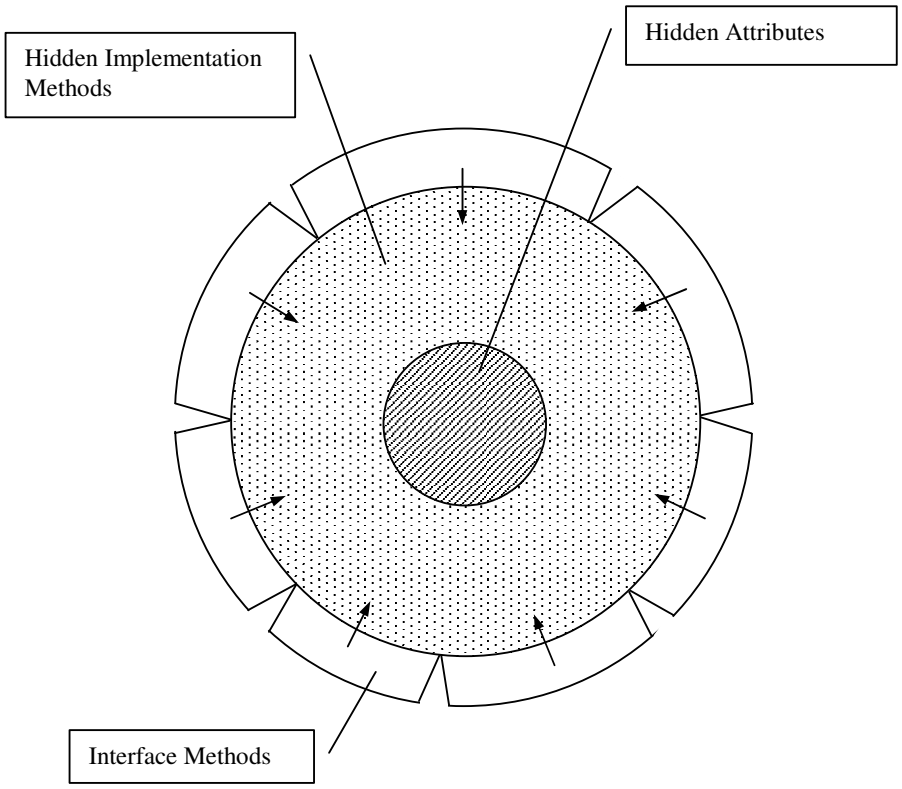


Figure 7-12 Representation of an object oriented module with low interface dependence

The measures of C++ class and object-class interface dependence contain sufficient information to enable the construction of a representation of the CPartFile module similar to that of Figure 7-12. This CPartFile representation can be compared to the ideal module represented by Figure 7-12 to identify ways in which the modularity of the CPartFile class and object-class can be improved by reducing their levels of interface dependence.

7.4.2.3.1 CPartFile class interface dependence

From Figure 3-7 of Chapter 3, the sub-characteristics of interface dependence that are directly described by the defined measures are 'interface size', 'data exposure' and 'interface element interdependence'. The following points list the measurement data required to construct a model of the CPartFile class similar to that of the ideal module represented by Figure 7-12.

- Interface Size
  - 0 interface attributes - CIS1:CCM1
  - 133 non-constructor and non-destructor interface methods - CIS3
  - 34 hidden attributes - CIS2
  - 5 hidden methods - CIS4

- Data Exposure
  - 0 interface attributes - CDE1:CCM1
  - 133 (all) interface methods directly access an attribute - CDE2:CCM2, CDE3:CCM3
  - 34 (all) hidden attributes directly accessed by an interface method - CDE4, CDE5

- Interface Element Interdependence
  - no interface attributes so no interface methods access interface attributes - CIEI1-6
  - 54 interface methods directly or indirectly invoke an interface method - CIEI7, CIEI8

Figure 7-13 represents the interface dependence of the CPartFile class. This figure is developed based on the same theoretical basis of low interface dependence that was used to develop the Figure 7-13 representation of a module with low interface dependence.

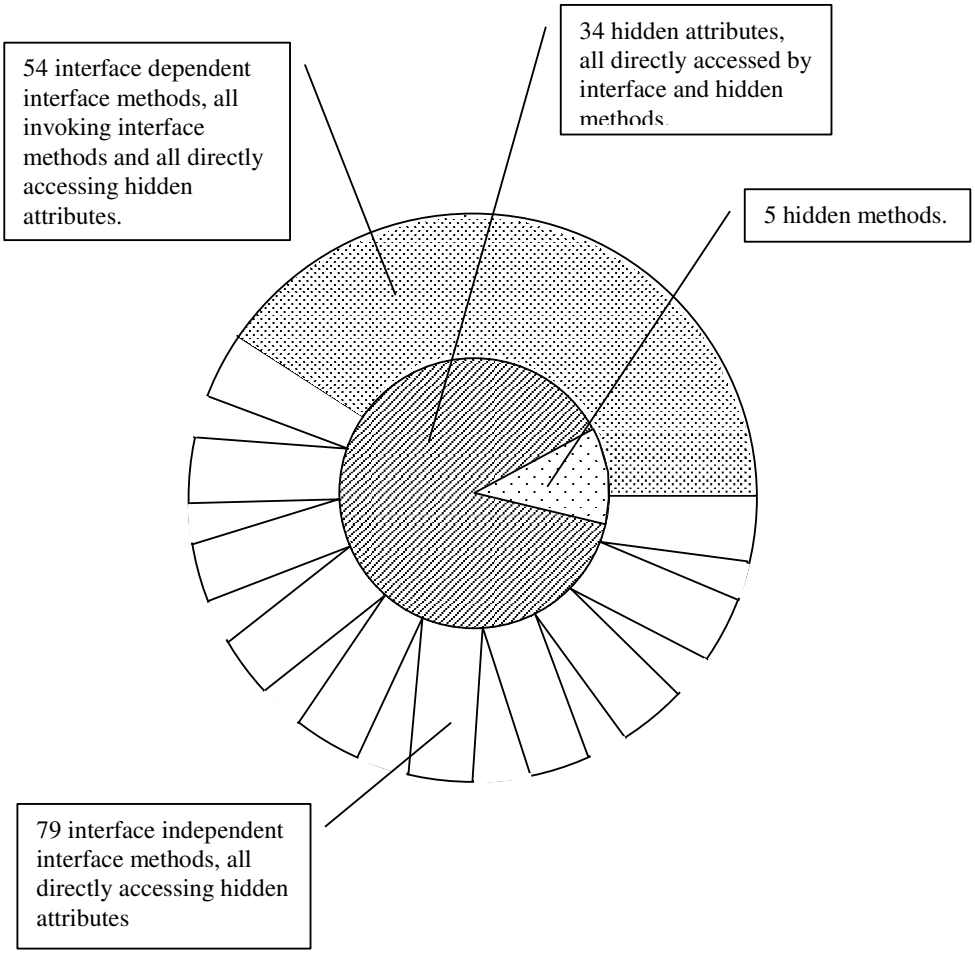


Figure 7-13 Graphical representation of CPartFile class interface dependence

Similarities between the Figure 7-12 ideal module and the Figure 7-13 CPartFile class representation of interface dependence are that:

- some CPartFile class interface methods do not invoke other CPartFile interface methods.
- the CPartFile class has no interface attributes.

Differences between the ideal module and the CPartFile class representation of interface dependence are that:

- the CPartFile interface contains a relatively high proportion of the total class methods while in the ideal module, the interface contains relatively few methods.
- 54 of the CPartFile class interface methods directly or indirectly invoke other class CPartFile interface methods whereas in the ideal module, no interface methods directly or indirectly invoke each other.
- all the CPartFile interface methods directly access one or more class attributes while in the ideal module, interface methods do not directly access class attributes.

These differences indicate that the interface dependence of the CPartFile class module could be decreased by:

- moving some of the class interface methods to the hidden part of the module.
- modifying the CPartFile class module implementation so that interface methods do not directly or indirectly invoke other interface methods. The services they are invoking should be provided by methods within the hidden part of the module.
- modifying the CPartFile class module implementation so that interface methods only indirectly access class attributes via hidden methods.

Point 1.2.3 of the natural language model of class interface dependence in Chapter 4, section 4.2.3.2.1 indicates that interface method lines of code, total method invocations and total attribute accesses describe the interface size and that interface size affects the level of interface dependence present in a module. Table 7-8 presents a summary of the CPartFile class interface method lines of code (CIS5), method invocations (CIS6), attribute reads (CIS7:CCM2) and attribute writes (CIS8:CCM3) measurement data. These measures can be used to identify the largest methods in the CPartFile class module that are most likely to contain implementation dependent code. Moving the methods thus identified to the hidden part of the CPartFile class module would reduce its interface dependence.

Class CPartFile Statistics

		CIS5	CIS6	CIS7:CCM2	CIS8:CCM3
N	Valid	133	133	133	133
	Missing	0	0	0	0
Mean		23.98	1.08	.59	.68
Median		12.00	.00	.00	.00
Mode		1	0	0	0
Std. Deviation		35.570	1.925	1.596	3.054
Skewness		3.124	2.718	4.562	8.619
Std. Error of Skewness		.210	.210	.210	.210
Kurtosis		12.788	8.601	28.353	85.453
Std. Error of Kurtosis		.417	.417	.417	.417
Range		228	11	13	32
Minimum		1	0	0	0
Maximum		229	11	13	32
Percentiles	25	1.00	.00	.00	.00
	50	12.00	.00	.00	.00
	75	32.50	1.50	.00	.00

Table 7-8 Summary of class CPartFile interface size measurement data

The information in Table 7-8 indicates that the distributions of the CPartFile CIS5-8 measured values are positively skewed. In this situation, the median value provides the most robust description of central tendency. Examination of the range and 75th percentile values for each CIS measure shows that each distribution as a few extreme outlier values. CPartFile methods with these extreme measured values are most likely to contribute to a decrease in CPartFile modularity.

Table 7-8 shows that some of the CPartFile class interface methods are relatively large having up to 229 lines of code, 11 method invocations and 32 direct attribute accesses. These relatively large interface methods are highly likely to contain implementation dependent code that could be moved to the hidden part of the class to decrease its level of interface dependence and hence improve its modularity. A measured CIS value greater than the 75th percentile value will be used to identify methods that are possibly increasing module CPartFile's interface size and hence reducing its modularity. The following analysis identifies some of these large interface methods.

Figure 7-14 shows the CPartFile class interface methods with lines of code greater than the CPartFile class 75th percentile value of 32.5.

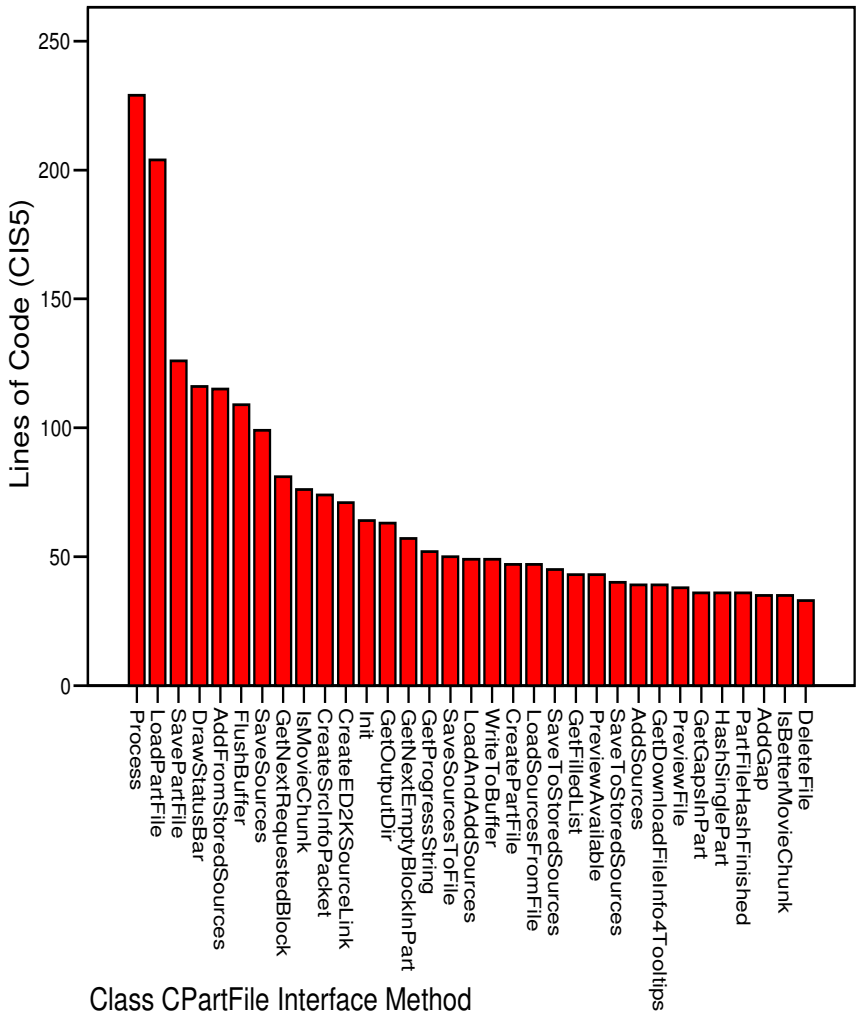


Figure 7-14 Class CPartFile interface methods with more than 32 lines of code

Figure 7-14 identifies CPartFile class interface methods Process and LoadPartFile as the most likely to contain service implementation specific code.

Figure 7-15 shows the cumulative value of measures CIS6 - methods directly invoked, CIS7:CCM2 - attributes directly read and CIS8:CCM3 - attributes directly written by CPartFile class interface methods. Only CPartFile class interface methods that invoke other methods or access attributes are included in this graph.

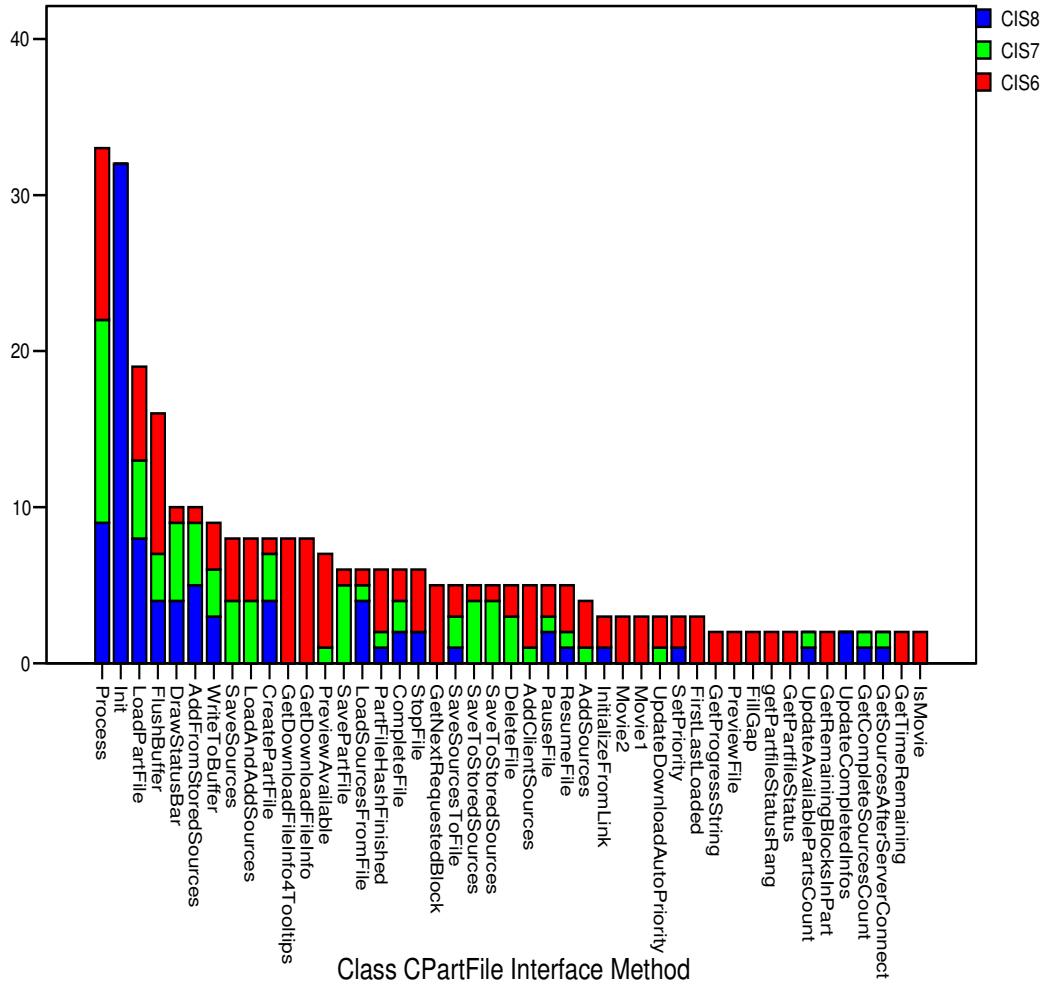


Figure 7-15 Class CPartFile interface method invocations and attribute accesses

Figure 7-15 identifies CPartFile class interface methods Process, Init, LoadPartFile and FlushBuffer as most likely to contain implementation specific code. Of these, the Init method is most likely to contain implementation specific code because measure CIS8:CCM3 indicates that it directly writes to 32 CPartFile class attributes.

Figure 7-16 plots the class CPartFile lines of code count vs. the sum of methods invoked and attributes accessed for interface methods indicated in both Figures 7-14 and 7-15 as potentially increasing the interface dependence of class CPartFile.

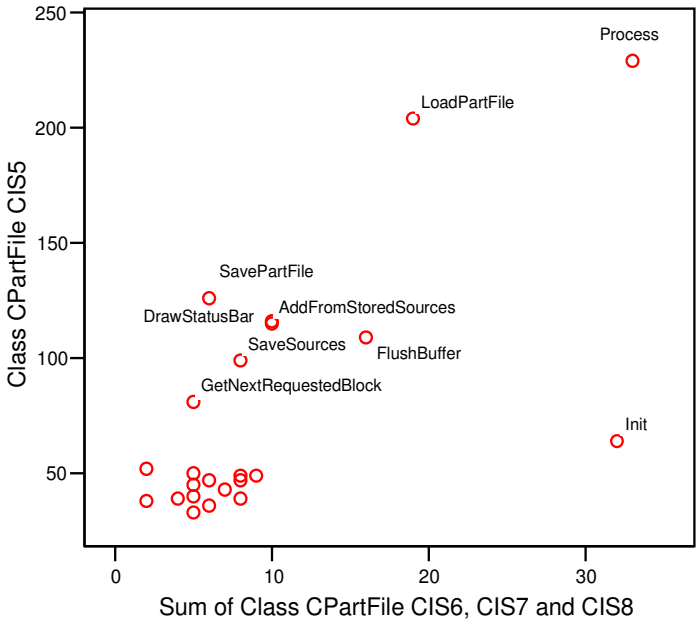


Figure 7-16 Class CPartFile interface methods that potentially increase interface dependence

The CPartFile interface methods annotated on the Figure 7-16 plot are relatively large compared to the other CPartFile class interface methods. These methods are indicated as being most likely to contain implementation dependent code and should be examined by hand to determine whether or not this is the case. If they do contain implementation specific code, then to reduce the interface dependence of the CPartFile class module, this code should be moved to methods within the hidden part of the module.

This case study aimed to answer the question "How can the interface dependence be changed to increase the modularity of the CPartFile class?". The points below summarise the previous discussion to answer this question

- move the code implementing method Init to the hidden part of the module.
- examine the Process, LoadPartFile, AddFromStoredSources, DrawStatusBar, SavePartFile, FlushBuffer, GetNextRequestedBlock and SaveSources interface methods to determine whether or not they contain implementation specific code. If they do, move this code to the hidden part of the module.
- modify the CPartFile implementation so that interface methods only directly invoke hidden methods and hidden methods do not invoke interface methods. This will ensure that interface methods are not themselves directly or indirectly dependent on other interface methods.
- modify the CPartFile implementation so that interface methods only indirectly access the class attributes via hidden methods.

These actions would help to reduce the levels of interface dependence of the CPartFile class and hence increase its modularity. It is important to remember that at system runtime, it is the CPartFile object rather than the class that is executing. For this reason, the interface dependence of the CPartFile object will affect the modularity of the eMulePlus system as it executes. The next section analyses and interprets CPartFile object interface measurement data to answer the question of "How can the CPartFile object interface dependence be changed to increase its modularity?"

7.4.2.3.2 CPartFile object interface dependence

The CPartFile object differs from the CPartFile class in two ways. Firstly, the CPartFile class interface is comprised of elements with public and protected levels of protection, while the CPartFile object interface is comprised of only elements with a public level of protection.

Secondly, the CPartFile object includes elements inherited from its one immediate parent class and two distant ancestor classes. Figure 7-17 illustrates the inheritance hierarchy of CPartFile.

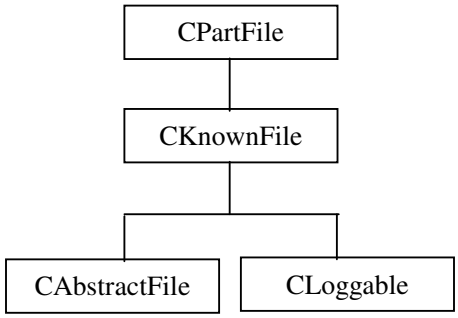


Figure 7-17 Object CPartFile inherits elements from three ancestor classes

The following analysis investigates ways in which CPartFile object interface dependence differs from CPartFile class interface dependence and how the interface dependence of the CPartFile object can be reduced to improve its level of modularity.

The following points list the measurement data required to construct a model of the CPartFile object similar to that of the ideal module represented by Figure 7-12.

- Interface Size
  - 4 interface attributes - OIS1:OCM1
  - 179 non-constructor and non-destructor interface methods - OIS3
  - 43 hidden attributes - OIS2
  - 16 hidden methods - OIS4

- Data Exposure
  - 4 interface attributes - ODE1:OCM1
  - 57 interface methods directly access an attribute - ODE2:OCM2, ODE3:OCM3
  - 122 interface methods do not directly access an attribute - ODE2:OCM2, ODE3:OCM3
  - 4 (all) interface attributes directly accessed by an interface method - ODE4, ODE5
  - 41 hidden attributes directly accessed by an interface method - ODE4, ODE5
  - 2 hidden attributes not directly accessed by interface methods - ODE4, ODE5

- Interface Element Interdependence
  - 25 interface methods directly or indirectly access interface attributes - OIEI1, OIEI2, OIEI4, OIEI5
  - 71 interface methods directly or indirectly invoke an interface method - OIEI7, OIEI8
  - 78 interface methods directly or indirectly access an interface element - OIEI1, OIEI2, OIEI4, OIEI5, OIEI7, OIEI8
  - 101 interface methods do not directly or indirectly access an interface element - OIEI1, OIEI2, OIEI4, OIEI5, OIEI7, OIEI8

Figure 7-18 represents the interface dependence of the CPartFile object. This figure is developed based on the same theoretical basis of low interface dependence that was used to develop the Figure 7-12 ideal module representation.

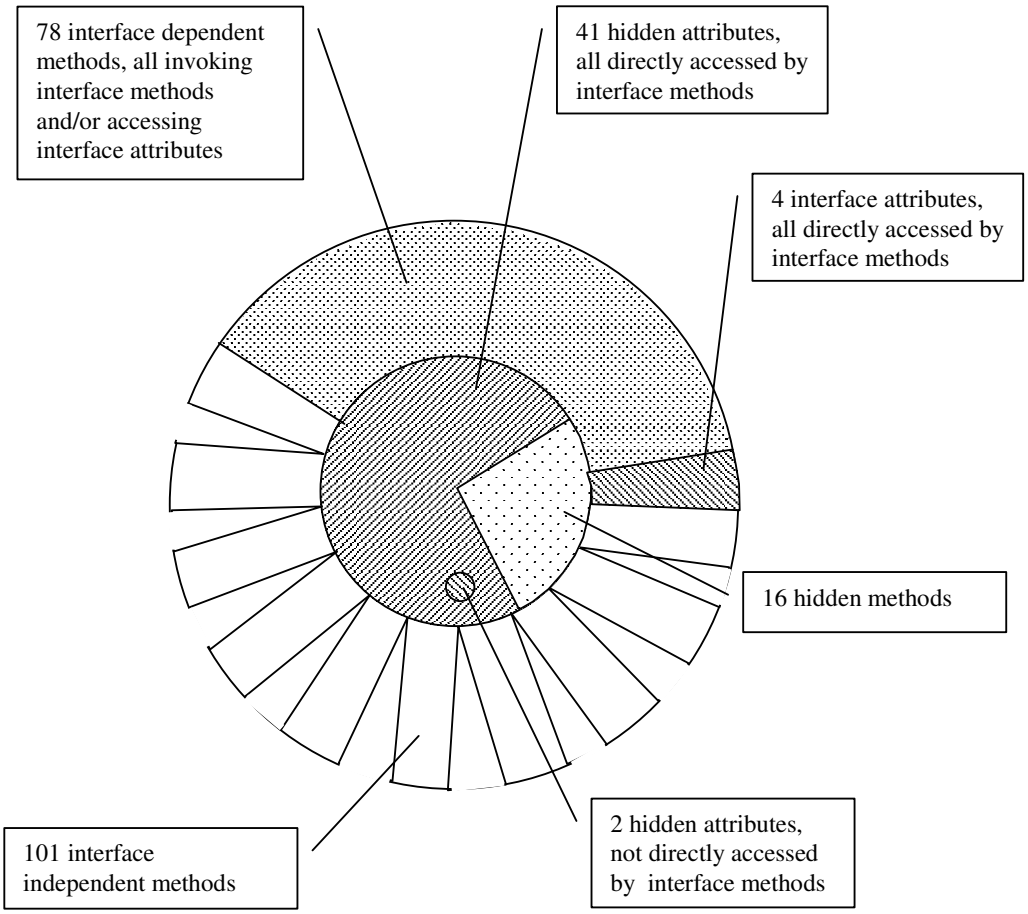


Figure 7-18 Graphical representation of CPartFile object interface dependence

Similarities between the Figure 7-12 ideal module and the Figure 7-18 CPartFile object representation of interface dependence are that:

- some CPartFile object interface methods do not invoke other CPartFile object interface methods.
- the CPartFile object has some hidden attributes that are not directly accessed by interface methods.

Differences between the ideal module and the CPartFile object representation of interface dependence are that:

- the CPartFile object interface contains a relatively high proportion of the total object methods while in the ideal module, the interface contains relatively few methods.
- 78 CPartFile object interface methods directly or indirectly invoke other CPartFile object interface methods whereas in the ideal module, no interface methods directly or indirectly invoke each other.
- 57 CPartFile object interface methods directly access a CPartFile object attribute and 122 interface methods do not. This is different to the ideal module where no interface methods access an attribute.
- the CPartFile object module has 4 interface attributes whereas the ideal module has none.

These differences indicate that the interface dependence of the CPartFile object module could be decreased by:

- moving some of the object interface methods to the hidden part of the module.
- moving all the object interface attributes to the hidden part of the module.
- modifying the CPartFile object module implementation so that interface methods do not directly or indirectly invoke other interface methods. The services they are invoking should be provided by methods within the hidden part of the module.
- modifying the CPartFile object module implementation so that interface methods only indirectly access object attributes via hidden methods.

Point 5.2.3 of the natural language model of object interface dependence in Chapter 4, section 4.2.4.2.1 indicates that interface method lines of code, total method invocations and total attribute accesses describe the interface size and that interface size affects the level of interface dependence present in a module. Table 7-9 presents a summary of the CPartFile object interface method lines of code (OIS5), method invocations (OIS6), attribute reads (OIS7:OCM2) and attribute writes (OIS8:OCM3) measurement data.

Object CPartFile Statistics

		OIS5	OIS6	OIS7:OCM2	OIS8:OCM3
N	Valid	179	179	179	179
	Missing	0	0	0	0
Mean		20.32	1.39	.66	.40
Median		9.00	.00	.00	.00
Mode		1	0	0	0
Std. Deviation		31.745	2.545	1.725	1.334
Skewness		3.539	2.910	4.566	5.198
Std. Error of Skewness		.182	.182	.182	.182
Kurtosis		16.890	9.979	26.211	32.538
Std. Error of Kurtosis		.361	.361	.361	.361
Range		228	14	14	11
Minimum		1	0	0	0
Maximum		229	14	14	11
Percentiles	25	1.00	.00	.00	.00
	50	9.00	.00	.00	.00
	75	28.00	2.00	1.00	.00

Table 7-9 Summary of object CPartFile interface size measurement data

The information in Table 7-9 indicates that the distributions of the CPartFile OIS5-8 measured values are positively skewed. In this situation, the median value provides the most robust description of central tendency. Examination of the range and 75th percentile values for each OIS measure shows that each distribution as a few extreme outlier values. CPartFile methods with these extreme measured values are most likely to contribute to a decrease in CPartFile object modularity.

Table 7-9 shows that some of the CPartFile object interface methods are relatively large having up to 229 lines of code, 14 method invocations and 25 direct attribute accesses. These relatively large interface methods are highly likely to contain implementation dependent code that could be moved to the hidden part of the class to decrease its level of interface dependence and hence improve its modularity. A measured OIS value greater than the 75th percentile value will be used to identify methods that are possibly increasing object module CPartFile's interface size and hence reducing its modularity. The following analysis identifies some of these large interface methods.

Figure 7-19 shows the CPartFile object interface methods with lines of code greater than the CPartFile object 75th percentile value of 28.

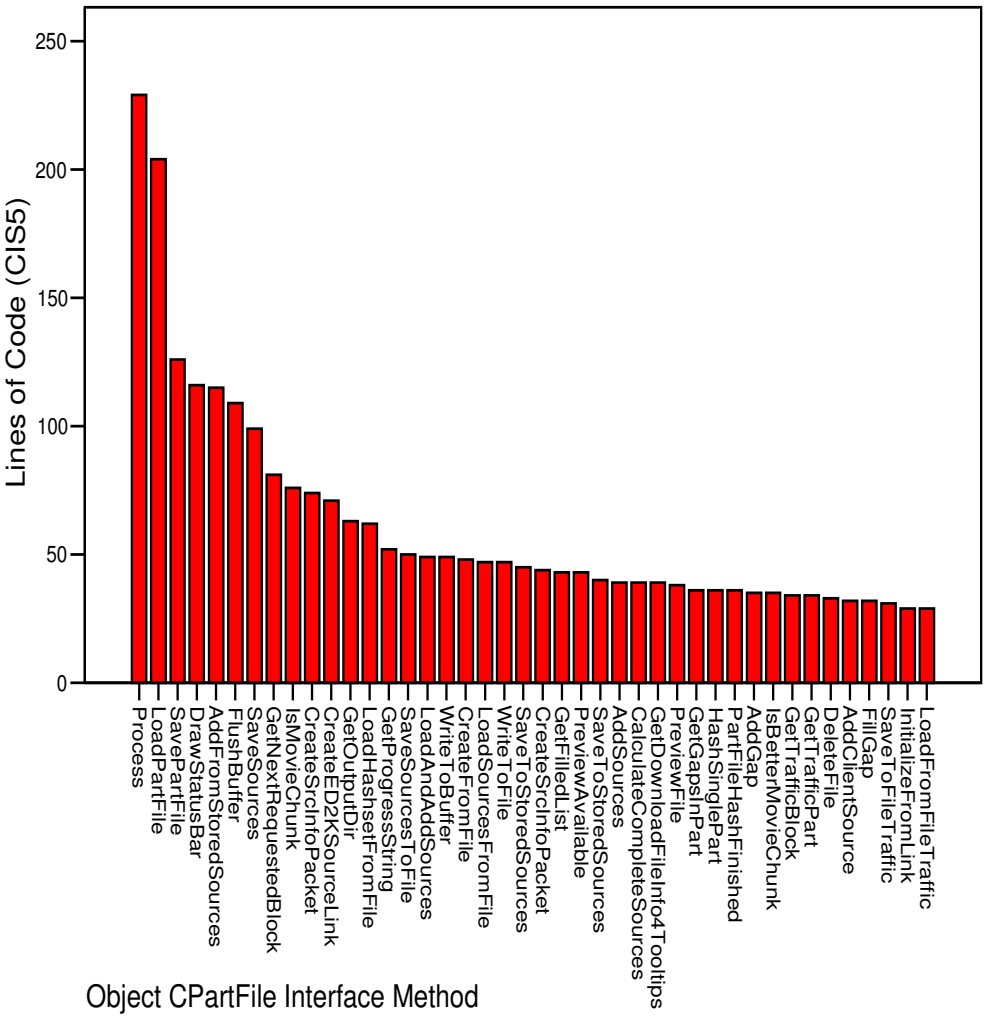


Figure 7-19 Object CPartFile interface methods with more than28 lines of code

Figure 7-19 identifies CPartFile object interface methods Process and LoadPartFile as most likely to contain service implementation code. These are the same methods identified in the CPartFile class interface dependence analysis as being most likely to contain implementation dependent code.

Figure 7-20 shows the cumulative value of measures OIS6 - methods directly invoked, OIS7:OCM2 - attributes directly read and OIS8:OCM3 - attributes directly written by CPartFile object interface methods. More CPartFile interface methods invoke other methods or access attributes than could be clearly displayed on a single graph. For this reason, in Figure 7-20, only CPartFile object interface methods that invoke more than one other method or access more than one attribute are included in this graph. Since this analysis is intended to identify interface methods that make a significant contribution to interface dependence, the omission of methods making only a slight contribution is acceptable.

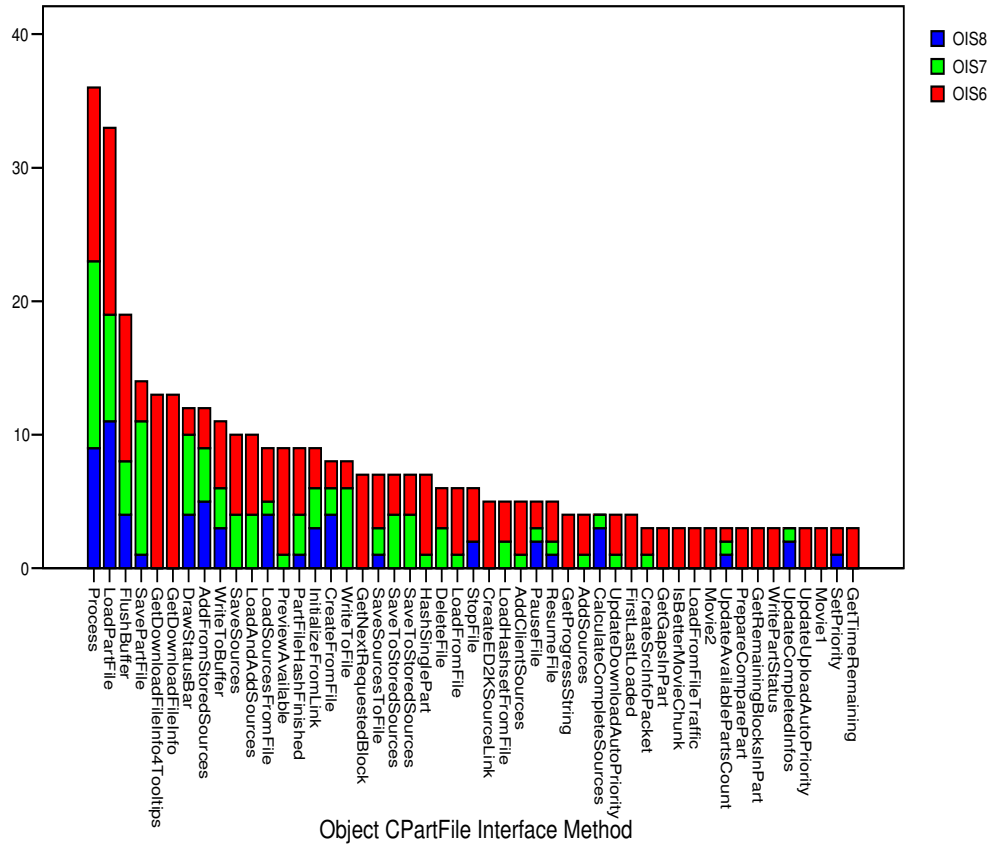


Figure 7-20 Object CPartFile interface method invocations and attribute accesses

Figure 7-20 identifies CPartFile object interface methods Process, LoadPartFile and FlushBuffer as most likely to contain implementation specific code. The Init method, identified in the CPartFile analysis as containing highly implementation dependent code, does not appear in the CPartFile object interface.

Figure 7-21 plots the CPartFile object lines of code count vs. the sum of methods invoked and attributes accessed for interface methods indicated in both Figures 7-19 and 7-20 as potentially increasing the interface dependence of the CPartFile object.

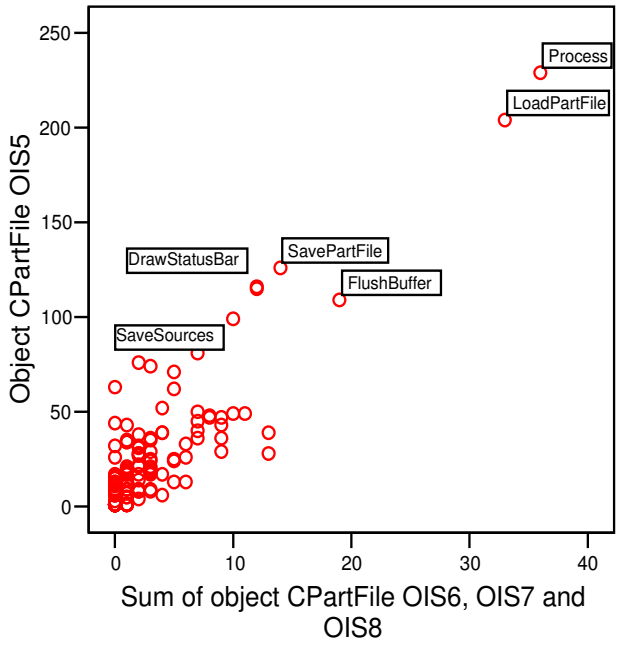


Figure 7-21 Object CPartFile interface methods that potentially increase interface dependence

The CPartFile interface methods annotated on the Figure 7-21 plot are relatively large compared to the other CPartFile object interface methods. These methods are indicated as being most likely to contain implementation dependent code and should be examined by hand to determine whether or not this is the case. If they do contain implementation specific code, then to reduce the interface dependence of the CPartFile object module, this code should be moved to methods within the hidden part of the module.

This case study aimed to answer the question "How can the interface dependence be changed to increase the modularity of the CPartFile object?". The points below summarise the previous discussion to answer this question

- examine the Process, LoadPartFile, AddFromStoredSources, DrawStatusBar, SavePartFile, FlushBuffer and SaveSources interface methods to determine whether or not they contain implementation specific code. If they do, move this code to the hidden part of the module.

- modify the CPartFile object implementation so that the 4 interface attributes are moved to the hidden part of the module.
- modify the CPartFile object implementation so that interface methods only directly invoke hidden methods and hidden methods do not invoke interface methods. This will ensure that interface methods are not themselves directly or indirectly dependent on other interface methods.
- modify the CPartFile object implementation so that interface methods only indirectly access the class attributes via hidden methods.
- examine the eMulePlus system to determine which CPartFile object interface methods are directly accessed by system elements external to CPartFile. Any interface methods not directly accessed can be moved to the hidden part of the object.

The CPartFile object is made up of more elements than the CPartFile class, and its interface is made up of only public elements, as opposed to the public and protected elements that comprise the CPartFile class interface. Despite these differences, many of the causes of their high levels of interface dependence are similar.

Similarities between the CPartFile class and object modules that increase their levels of interface dependence are that:

- both have a relatively large number of module methods appearing in their interface.
- both have a high proportion of interface methods directly or indirectly invoking other interface methods.
- for both the CPartFile class and object, the same interface methods are identified as most likely to contain implementation dependent code. The only exception to this is method Init, which appears in the CPartFile class interface but not in the object interface.
- both CPartFile class and object have interface methods that directly access module attributes.

Significant differences between CPartFile class and object interface dependence are:

- the CPartFile object has interface attributes while the CPartFile has none. In this way, CPartFile class interface dependence is less than that of the CPartFile object.
- all the 133 CPartFile class interface methods directly access attributes while only 57 of the 179 CPartFile object interface methods directly access attributes. In this way, CPartFile object interface dependence is less than that of the CPartFile class.

This case study shows that reducing the interface dependence of the CPartFile class module in the ways outlined in section 7.2.4.1, would have the effect of also significantly reducing the interface dependence of the CPartFile object module. Additional measures needed to further reduce the CPartFile object interface dependence are the removal to the hidden part of the module of the 4 interface attributes and any interface methods not invoked by external eMulePlus system elements. Reducing the levels of interface dependence in these ways would increase the modularity of the CPartFile class and object modules.

7.4.2.4 Discussion

The natural language entity model of class and object interface dependence contains sufficient detail such that a graphical representation of an ideal module with a low level of interface dependence, as shown in Figure 7-12, can be constructed. The measures developed from this natural language entity model provide a detailed description of C++ class and object interface dependence that can be used to construct a graphical representation of a measured module. This is a powerful method of analysis capable of presenting information from almost 2000 data points in a relatively easily understood way. Figures 7-13 and 7-18 graphically represent the interface dependence of the CPartFile class and object modules. The interface dependence of these measured CPartFile modules can be interpreted with respect to the ideal module by comparing their images. Differences between the measured and ideal module images indicate ways in which the measured modules can be modified to improve their levels of interface dependence. Guided by this information, detailed analysis of individual measures can be performed to identify specific ways in which the implementation of the CPartFile class and object modules can be changed to reduce their levels of interface dependence and hence increase their modularity.

7.5. Conclusions

Content validation is an important preliminary step to applying a measurement instrument to a software system. Content validation aims to demonstrate that the measures implemented within the measurement instrument provide an adequate description of the software system. The level of content validity of a set of implemented measures is dependent on the particular software features possessed by the software system to be measured and the particular aim of the user in applying the measures to the software system.

The descriptive software measure development process described and demonstrated in this thesis facilitates content validation by maintaining the links between the characteristic and sub-characteristics of interest, the software features that affect the levels of these characteristics and sub-characteristics present in the software, and the measures defined to quantify these features. The CHARMER diagrams express these connections. The CHARMER diagrams are also able to express conditional links between characteristics, features and measures. Investigating whether these special conditions apply to the software system to be measures forms part of the content validation. Finally, CHARMER diagrams are able to indicate software features that are not, for some reason, described by implemented measures. As demonstrated in case study 1, section 7.1.2, the information contained within the CHARMER diagrams, combined with the importance and frequency ratings specified by a potential user, can be used directly to facilitate content validation.

Although it is important to ensure that measures have content validity, the process of determining content validity is subjective and content validity alone is not "fully sufficient for assessing the validity" (Carmines and Zeller 1979, p. 22) of measures of complex characteristics such as software modularity. Construct validity offers an objective estimate of measure validity. "Construct validity is … central to the measurement of abstract theoretical concepts." (Carmines and Zeller 1979, p. 23) The simple construct validation presented in this chapter provided evidence of the construct validity of the defined measures of C++ class modularity when they were combined into unweighted and weighted aggregate values and into a Euclidean distance value. To be able to generally accept the construct validity of the measures of C++ class and object modularity, further work is needed. This work should estimate construct validity for different software system validation cases against multiple measures of several criteria.

If the validity of measures, as they are implemented in a measurement instrument, is determined to be sufficient for the intended measurement purpose, the instrument can be applied to the software system and measurement data extracted. The resulting data set can be relatively large, requiring appropriate analysis to derive the required information. The aim in performing the measurement will in part dictate the appropriate analysis technique. Having analysed the data, it must be interpreted. The sub-characterisation and conceptual definition of the characteristic of interest and the natural language entity model express the theoretical basis from which the measures were developed. The measured data is interpreted with respect to this theoretical basis, clearly identified by the CHARMER diagrams developed as part of the measure development process.

The case studies performed in this chapter demonstrate the content validation of the set of modularity measures with respect to the eMulePlus software system. The first case study showed how the modularity measurement data obtained from the eMulePlus system could be aggregated to provide a general description of class and object modularity. The modularity sub-characterisation described in Chapter 3 provided the theoretical basis for this analysis. The second case study showed how the modularity measurement data could be used to provide a detailed description of the interface dependence of a single module.

The systematic method of software descriptive measure development illustrated in Figure 1-3 and described and demonstrated in Chapters 3, 4, 5 and 6 can facilitate the production of a set of measures that can be implemented, validated, applied to a measurement task and the resulting measured data analysed and interpreted. Future work is needed to demonstrate the wider applicability of the measurement process to the development of descriptive measures of software
characteristics other than modularity. Future work is also suggested to investigate the usefulness of the C++ class and object modularity measures developed and demonstrated as part of this thesis.

8. Discussion and Suggestions for Further Work

This thesis has shown that, where a software characteristic is sufficiently well understood, the systematic process of descriptive measure development illustrated in Figure 1-3 can be used to define and implement a set of measures that provide a detailed description of this characteristic. The systematic measure development process supports the development of conceptual definitions, natural language and mathematical entity models and measure definitions, which establish the explicit descriptive measurement relationships shown in Figure 1-8. These explicit relationships create the implicit descriptive measurement relationship between the characteristic of interest and the defined measures, allowing the measures to describe the characteristic. By assessing these explicit relationships, summarised in the CHARMER diagram, a user is able to perform a content type validation of the measures with respect to an intended application. Where the theoretical basis from which the measures were developed identifies other characteristics related to the measured characteristic of interest, and where valid measures of these other characteristics exist, construct validation can also be carried out. If the measures are judged to have sufficient validity, and are applied to a software system, the systematic measure development process provides sufficient information, again summarised in the CHARMER diagram, to allow the user to analyse and interpret the resulting measurement data.

8.1. Research Outcomes

There are several outcomes of the research described in this thesis document. First and foremost is the systematic process of software descriptive measure development. This process represents the most significant outcome because it is applicable to the development of measures to describe software characteristics other than the modularity characteristic for which measures were developed here. These measures describing object oriented C++ class and object modularity are the second outcome of this research. The final significant outcome of the research is the case study performed on the eMulePlus software system. This case study is significant because it demonstrates validation of the measures as well as analysis and interpretation of the measurement data.

The following sections discuss each of these research outcomes in terms of both the positive contributions they make to the field of software engineering, and in terms of any recognised limitations or shortcomings.

8.1.1 Systematic process of software descriptive measure development

The systematic process of software descriptive measure development described in section 1.3 and demonstrated in this thesis document supports the definition, implementation, validation and analysis and interpretation of measures to describe software characteristics of interest. This process emphasises the need to complete each stage of the process before moving to the next stage. The systematic measure development process specifies the different products that together define a set of measures, and identifies and describes the theoretical basis from which the measures were derived. The process also specifies the order in which these products should be produced. This is to ensure that sufficient information is documented in previous stages to complete the following stage. The progression through the measure development stages is such that a documented path is created between each characteristic and sub-characteristic of interest via the software features identified as affecting these characteristics, to the individual measures quantifying the features and hence describing the characteristics. This path, summarised in the CHARMER diagram, supports the content validation of the measures and later supports the analysis and interpretation of the measurement data.

While contributing to the rigour with which software descriptive measures are defined, the systematic process of descriptive measure development has a number of currently identified limitations. The first of these is that it is necessary to begin the process with sufficient theoretical understanding of the characteristic to be described by the measures to be able to conceptually define it, including a sub-characterisation if necessary. This means that the process is applicable to the development of measures to describe characteristics that are considered by the measure developer to be sufficiently well understood. The judgement as to whether or not the available theoretical understanding of a characteristic is sufficient for measure development should be based on the intended purpose of the measures. A limited understanding will result in measures providing a less detailed, and possibly less valid, description of the software, while a more detailed understanding will result in a more detailed and possibly more valid description.

While the systematic process of software descriptive measure development does not specifically include mechanisms to identify ways in which the theoretical basis of defined measures can be improved, it does indirectly support the development of such theory. In the first place, the systematic process of measure development indicates the type of theoretical understanding needed to support the development of descriptive measures. To complete the

characteristic conceptual definition stage of measure development, theory relating to the sub-characterisation of complex characteristics is required, as well as sufficient theoretical understanding of the characteristic and sub-characteristics to be able to define them conceptually in relatively simple terms. To complete the natural language modelling of the software, a theoretical understanding of the features of the software that affect the levels of characteristic and sub-characteristic present is required. Finally, to perform a construct validation, an understanding of characteristics that are related to the characteristic of interest is required. Research effort can be focussed on developing these types of theoretical understanding that will then support the development of software descriptive measures. Documenting the theoretical basis of the developed measures in the conceptual definition and natural language model of the characteristic allows future measure researchers to examine this theory and enhance it with new information as it becomes available. For example, research regarding a particular software feature such as inheritance may identify new ways in which it affects modularity of objects. Examination of a natural language model of object modularity may show that this feature is not included. The measures developed based on this model may be improved by adding the new feature to the model and defining new measures to quantify it.

Explicitly describing the theoretical basis from which the measures are developed is important as this information supports the validation and interpretation of the measurement data. As shown in Figure 1-1 of Chapter 1, the theoretical basis for measure development identifies software features affecting characteristics of interest. The execution of the systematic measure development process is such that characteristics and sub-characteristics of interest are first identified, followed by the identification of the software features that affect the levels of these characteristics and sub-characteristics in the software. Where no features can be identified for a particular characteristic, this is recorded in the documentation of the measure development process. The opposite situation of identifying software features that are not understood sufficiently to be able to say whether or not they affect the characteristic of interest is not recorded as part of the systematic measure development process. In its current form, the systematic measure development process does not identify software features that have not been included in the natural language model because the measure developer could not find any theory describing whether or not the feature contributed to the levels of characteristic in the software. For example, the C++ language supports the definition of class templates. As no theory was discovered to describe whether or not this software feature affects modularity, it was omitted from the natural language model. Such an omission implies that templates have little or no affect on class modularity where as the truth was that there was no information

available to determine the effect of templates on modularity. While omitted features do not belong in the natural language model, another document could be included in the systematic measure development process identifying features whose effect on the characteristic of interest is unknown. Should a more detailed description of a characteristic be required, this list could form a basis for future research to identify new ways that features affect the characteristic of interest.

An important consideration, once the measures have been developed, is that their validity be established in some way. Validation demonstrates the degree to which a measure or set of measures is describing the characteristic they purport to (Carmines and Zeller 1979, p. 12). Such validation builds evidence as to the validity of measures rather than proving it conclusively. The systematic process of measure development advocates content validity as an appropriate form of validation of the measure definition products. This is because content validation is based only on the theory that was used to develop the measures in the first place. No extra theory or measures are needed. The main limitations of content validity are that it is a subjective process based on the judgement of the person examining the measures. It is also based on the examination and acceptance of the theoretical basis from which the measures were developed. This theory is not tested objectively by content validation. Another type of validation appropriate to the descriptive measures developed according to the systematic process is construct validation. This validation involves the objective assessment of the degree to which measures of a characteristics are related to measures of another characteristic theoretically related to the first. Because construct validation involves theory and measures beyond those of the systematic measure development of a particular characteristic of interest, it is not explicitly included in the development process. This is not to say that construct validation is not important or necessary, but rather it is saying that it is a separate process to be executed after the systematic measure development has occurred. An area for possible future work is the development of a process of construct validation that can be added on to the existing measure development process.

Another shortcoming of the systematic measure development process is that, for complex software characteristics, it is not simple to perform. If all that is required of a measure is that it indicates cases with potentially unacceptable levels of a particular characteristic, then a single measure, simply defined, may be adequate. For example, the construct validation in section 7-3 demonstrated that, for the eMulePlus software system, a simple class lines-of-code count could be used as a moderately valid indicator of class modularity. The difference between using the

lines of code measure and the full set of modularity measures is that the lines of code count is only able to indicate that modularity could be improved by reducing the lines of code of a class. On the other hand, the modularity measures identify the specific class features that cause the high modularity, allowing targeting of software improvement effort. One way to determine whether or not the effort in developing a set of measures of a particular software characteristic according to the systematic process of measure development is justified is to consider how widely applicable and useful the measures will be. For example, modularity is a well recognised characteristic of high quality software and measures describing modularity have the potential to be widely useful, justifying the effort needed to develop them. By contrast, a set of measures providing a detailed description of a more obscure and less relevant software characteristic may not justify the effort spent developing them. As for any development activity, the perceived benefit must balance the effort needed to create the product.

An issue regarding the systematic measure development process is that it advocates the use of a mathematical model of the software to support mathematical measure definitions. Ideally, measurement data is collected from the features that have the most significant effect on the characteristic of interest, while less significant features are not measured. The use of a mathematical model may have the unintended effect of altering the features from which measurement data is collected from the most significant, to those that can be most easily included in the selected mathematical model. To recognise the possible limitations of mathematical measure definitions, natural language definitions should be given for measures quantifying all the software features identified in the natural language software model. After this, where possible, mathematical measure definitions should also be given. By providing both natural language and mathematical measure definitions, features affecting the characteristic of interest that are included in the natural language model but not in the mathematical model will not be overlooked. The measures of the object connection obscurity sub-characteristic of connection via non-standard interface given in Tables 5-26 and 5-27 of section 5.2.4.3 show examples of measures defined in natural language terms that cannot then be defined in terms of the implemented mathematical model. The natural language measure definitions are independent of the selected mathematical software model and so can be adapted to different software models. In this thesis, the mathematical model and measure definitions are implementation dependent. It is possible that a generic, implementation independent mathematical model can be defined and similarly implementation independent measures defined from it. This is a potential area for future research. It is important to note however that

implementation independent mathematical models and measure definitions must eventually be defined in implementation dependent terms so that a measurement instrument can be constructed.

Another issue for discussion regarding the systematic process of measure development is that the measures are only classified according to the particular sub-characteristics and characteristic they describe. While other classification schemes are possible, the process as defined in this thesis does not identify or accommodate them. For example, measures quantifying features that theory suggests have a strong effect on the characteristic of interest could be rated more highly than those quantifying features believed to have a lesser effect. These ratings could be used in the analysis phase to highlight particularly important measures. Another possible classification scheme could be according to the level of description provided by each measure. Some measures may provide a general view of the software characteristic while others may be describing the software at a more detailed level. Such identification of the level of detail provided by each measure is according to the theory from which the measures were derived. An area for future research is the ways in which the measures can be classified and how these classifications can be used to aid the analysis and interpretation of the measurement data.

The systematic process of software descriptive measure development described and demonstrated in this thesis is capable of supporting the development of measures that provide a detailed description of a complex software characteristic. While the execution of this process requires thought and effort on the part of the measure developer, and may require several iterations to achieve a satisfactory result, it has been shown in the thesis to produce measures that can be successfully validated, analysed and interpreted to provide an understanding of the levels of modularity present in a software system. It is not contended that this process is fully refined and ready to be widely promoted to the software development community however, it is contended that it provides a useful means of defining descriptive measures that warrants further investigation and improvement.

8.1.2 Descriptive measures of C++ class and object modularity

The set of measures describing C++ class and object modularity represent another important outcome of the research described in this thesis. These measures provide a detailed description of the complex software characteristic of modularity. This description is derived from the

theoretical basis of Meyer's (1997 pp. 46-53) five rules of modularity. This theory is interpreted with reference to C++ class and object modules and this interpretation is illustrated in the diagrammatic conceptual models of modularity of Figures 3-3, 3-4, 3-5 and 3-6 in Chapter 3 and described in the conceptual definitions, and the associated natural language models of modularity. The sub-characteristics selected to describe modularity are defined such that an absence of the sub-characteristic, indicated by a measured value of zero, indicates optimum modularity. This is intended to simplify the analysis of the measurement data. Measures are defined to quantify the features identified in the natural language model as reducing the modularity of a C++ class or object module.

In the set of measures defined to describe C++ class and object modularity, some features of C++ software are associated with more than one modularity sub-characteristic. These features are identified and discussed in section 4.2.3.6 of Chapter 4. While having several measures quantifying the same feature provides a natural weighting in the final measured description to these more widely influential features, it also introduces dependencies between the measured descriptions of sub-characteristics sharing common measured features. These dependencies may cause problems when using an analysis technique based on the independence of the measured description of separate software characteristics and sub-characteristics. When analysing such measured data, one approach could be to remove unwanted dependencies by not using the data from measures describing common features. To facilitate this, Table 5-14 and Table 5-32 in Chapter 5 identifies measures describing common C++ class and object software features.

In the set of measures describing of C++ class and object modularity, some sub-characteristics are described by several measures while others are described by only a few. The sub-characteristics described by many measures may have a more significant effect on the overall description of modularity that the sub-characteristics described by a few measures only because of this extra data rather than because the sub-characteristic is more significant. When analysing the measurement data, it may be necessary to equalise the contribution that each sub-characteristic makes to the final measured description. For example, in the calculation of the modularity aggregate value described in section 7.1.1, progressive summation and normalisation of measures describing each modularity sub-characteristic ensures that each sub-characteristic measured description contributes to the final modularity measured description according to its significance in the sub-characteristic hierarchy. By contrast, in calculating the modularity indicator using Euclidean distance, as described in section 7.1.2, all measures have

equal significance which means that sub-characteristics described by many measures have a greater representation, and hence greater effect on the final modularity value than those described by few measures.

Validation of the defined measures of modularity is important as the use of invalid measures could have significant negative consequences for a software development project. As with any measure, before it should be used, it is important to demonstrate that the measure or set of measures provide an adequate description of the characteristic they purport to. Validation is performed to demonstrate this. In this thesis, content and construct validation are demonstrated. Content validation, described in section 7.2, is a subjective type of validation that involves examination of the theory linking the characteristic of interest to the measured software features and examination of the measures quantifying these features. Some drawbacks of content validation include that the software system to be measured must be examined prior to performing the content validation and that the content validity assessment for a particular software system is only applicable to that system and should not be generalised to other software systems. A positive aspect of content validation is that the theoretical understanding needed to perform it is described within the products of the systematic measure development process.

Construct validation, described in section 7-3, is an objective type of validation that involves examining the relationships between the measures of the characteristic of interest and measures of several other characteristics that theory suggests are related. Evidence of a relationship between the measures describing the characteristic of interest and a single measure of a single theoretically related criterion is not sufficient to demonstrate the construct validity of the measures. To be able to declare measures to have a general level of acceptable construct validity, it is necessary to perform many construct validations against several different measures describing several different theoretically related criteria for many different software systems. A problem that could be encountered when attempting to demonstrate the construct validity of measures is that the theoretical understanding of the characteristic may not be sufficient to identify other theoretically related criterion characteristics. Carmines and Zeller, in discussing the construct validation of complex characteristics in the field of social science, note that it is not true “that only formal, fully developed theories are relevant to construct validation….What is required is that one be able to state several theoretically derived hypotheses involving the particular concept.” (Carmines & Zeller 1979, pp. 23-24). Once sufficient construct validations have been performed, and the theoretical relationships between

the characteristic of interest and the criteria characteristics are all supported by the construct validation evidence, then the measures of the characteristic of interest can be declared to have sufficient construct validity in the generally sense. The full construct validation of the measures of modularity developed in this thesis is an area for future research as it will require a lot of time and effort to perform thoroughly and correctly.

Having defined the set of measures to describe C++ class and object modularity, and demonstrated content and construct validity against the eMulePlus software system, another outcome of this research is the case study describing the modularity of the eMulePlus software system.

8.1.3 Case study – eMulePlus software system

The case study described in section 7.4 of Chapter 7 demonstrated the analysis and interpretation of the modularity measurement data. The first case study used the modularity aggregate calculation to provide a general description of the modularity of all the measured eMulePlus classes and objects. The content validation performed on the measures before they were used in the case study showed that while the class modularity measured had sufficient validity to describe the eMulePlus system, the object modularity measures did not. The case study demonstrated one approach to the problem of reduced content validity by presenting a partial description of object modularity and recognising the incomplete description in the interpretation. Strengths of using an aggregation of the measures to represent modularity as a single value are that the relative modularity of individual classes and object within an entire system can be represented on a graph such as a scatter plot. This makes it easier to identify modules with particularly poor modularity that could benefit from remedial work. A drawback of representing modularity as a single value is that much of the descriptive detailed contained in the measures is lost. While the aggregate value provides a high level description that is suitable for identifying cases with relatively low, medium or high modularity, it is not suitable for identifying how the modularity of each case can be changed. For this, the individual measures must be examined. The second case study demonstrated the analysis of individual measures to provide a detailed description of a class and object identified as having low modularity.

In the case study, the measured values were used to construct a conceptual illustration of the structure of the selected class and object that represented its interface dependence. This

representation was compared to a similar representation of a module with high modularity, allowing the differences between the modules to be observed. This conceptual diagrammatic representation provided a way to present a detailed description of the module in a comprehensible form. A drawback of this type of data analysis is that presenting the entire measured system in this way could overwhelm the viewer with too much information. To overcome this problem, a high level overview description of the whole system could be presented, with the ability to allow the viewer to zoom in on parts of the system representation for a more detailed description. This is a possible avenue for future work.

The two forms of data analysis presented in the case study of the eMulePlus system were shown to be compatible, as the aggregate calculation presented an overview of relative modularity of the classes and objects, and the interface dependence conceptual representation provided a sufficiently detailed description of a module that specific ways to improve the modularity of individual modules could be determined.

8.1.4 Summary – research contributions and considerations

The following points summarise the main contributions made by this research and identify some considerations that may prove to be obstacles to the adoption of the systematic measure development process in the wider software engineering community.

8.1.4.1 Contributions

- Defined and demonstrated a systematic process by which software descriptive measures can be developed, implemented, validated, applied and analysed and interpreted. The measure development process promotes the explicit documentation of the theoretical basis from which the measures were developed and the relationships between software characteristic and sub-characteristic, through software feature to the measures that quantify the software feature and hence describe a particular characteristic or sub-characteristic of interest.
- Defined, implemented and performed preliminary validation of a set of measures to describe modularity of C++ class and object modules. These measures provide a detailed description of modularity that can be used to identify explicit ways in which modularity can be improved.

- Defined and demonstrated a data analysis technique involving the calculation of a single modularity aggregate value to summarise the modularity of C++ classes and objects. Performed an example construct validation of the modularity aggregate against a simple module size measure. This validation provided evidence of a moderate degree of construct validity.
- A case study of the eMulePlus software system to describe various aspects of its modularity. A general description of the levels of class and object modularity of the eMulePlus system was obtained from the calculation of a modularity aggregation. From this, identified a class and object-class with relatively low modularity. Performed a detailed analysis of the levels of interface dependence of this class and identified ways in which its modularity could be improved.

8.1.4.2 Considerations

- A prerequisite of using the systematic process of measure development is a theoretical understanding of the ways in which features of the software affect the levels of characteristic of interest present in the software. A detailed understanding will support the development of measures to provide a detailed description. A less detailed understanding will result in less detailed measures.
- Ideally, objective construct validation is used to demonstrate that the developed measures are sufficiently valid. The process of construct validation is based on a theoretical understanding of the measured characteristic. Where this understanding is limited, such as in the relatively new field of software engineering, this may limit the extent to which construct validation can be performed. Content validation is a type of validation that is also important, however it is a subjective type of validation that is specific to each measured software system. Performing a content validation is better than doing no validation at all however ideally, both content and construct validity is established before measures are used.
- The systematic measure development process is long and detailed, and time consuming to perform. The software characteristics to which the process is most applicable are those that are of wide interest to the software engineering community, and that have the potential for many applications since this will justify the development time and effort.

- Currently, the C++ class and object modularity mathematical software model and measure definitions are defined in an implementation specific way. The particular mathematical model chosen limited the number of measures that could be mathematically defined. Since the natural language model and natural language measure definitions are not implementation specific, necessary information is retained. It is possible that a less implementation specific mathematical model would have supported the mathematical definition of more measures. Ultimately, the mathematical model and measures will be defined in an implementation specific way so that they can be included in a measurement instrument implementation, however delaying this step could be advantageous.
- Only a single software system was examined in the case study. This system contained a significant number of measured classes and object-classes which provided a good demonstration of the modularity measures. This is a positive aspect of the case study. It is not possible to predict whether the eMulePlus software system examined in the case study is representative of C++ software systems in general. The validation and data analysis results obtained from the eMulePlus system should not be generalised for all software systems. The modularity measures should be applied to many diverse software systems before they are more widely accepted and used.

8.1.5 Further work

While this thesis has successfully shown that it is possible to develop descriptive software measures according to a systematic process, it has also identified several areas in which further work could be performed. The following points list some areas for further work.

- With regard to the systematic process of descriptive measure development defined in this thesis, an avenue for further work would be to use this process to develop a set of measures describing another complex software characteristic. By developing another set of measures, the systematic process will be further tested and ways in which it could be improved, identified.
- It is not possible to use the systematic process of measure development defined in this thesis without a good understanding of the characteristic of interest. Further work could be performed to sub-characterise and conceptually define some of the complex software characteristics that are of interest to software engineers. Examples of these include coupling, cohesion and complexity.

- Another prerequisite to the systematic measure development process defined in this thesis is a good understanding of the ways in which the conceptually defined characteristics and sub-characteristics are manifest in the software. Once complex software characteristics have been conceptually defined, further work could be performed to identify the features of software that affect their levels within the software.
- With regard to the mathematical model of C++ class and object modularity developed in this thesis, further work could be performed to find an alternative model capable of describing all the features identified in the natural language modularity model. A better model would support the definition of measures to provide a more detailed description of modularity. This model may need to be conceptually defined rather than implementation specific, as the one in this thesis is.
- With regard to the implementation of the modularity measures, further work could be done to identify a measurement instrument platform capable of implementing all the defined measures and collecting measurement data from larger software systems than the eMulePlus system used in the case study.
- With regard to the validation of the modularity measures, further work could be performed to demonstrate different types of validity. For example, if the content validity of the modularity measures defined in this thesis is accepted as being sufficient, then Meyer's theory of modularity could form the basis of a construct type validation. This construct validation (Diamantopoulos & Schlegelmilch 1997, p. 35) would demonstrate the degree to which the modularity descriptive measures, defined in this thesis based on Meyer's rules of modularity (Meyer 1997, pp. 46-53), are related to Meyer's five criteria of modularity (Meyer 1997, pp. 40-46). While a simple construct validation was demonstrated in the case study, this is not sufficient to declare the measures to have a generally acceptable level of construct validity. Validation of the measures against several different measures of several criterion variables for several software systems would be necessary to build evidence of construct validity.
- Another avenue for further work is to investigate other ways to analyse the measurement data obtained from the modularity measures defined in this thesis. One possibility is to identify the measures describing aspects of modularity that have a large influence on the overall levels of modularity present in the software. The reduced set of measures thus identified could be an alternative to the full measurement set when a less detailed description of the software modularity is required.

- Methods of presenting the measurement data in a way that is easily interpreted is another avenue for future work. One potential avenue of investigation is the interactive, software based visual presentation of a high level software system description that can be selectively refined and zoomed in on where the viewer chooses to ask for more information.
- The measures defined in this thesis describe C++ class and object modularity. Java is a popular object oriented programming language that is very similar to C++. Further work could be performed to modify the C++ measures developed in this thesis to describe Java class and object modularity.
- This thesis defined and applied a process of measure development to the task of developing descriptive measures of software products. It is possible that this systematic method of measure development could be modified to support the definition of descriptive measures of characteristics related to software processes.

Software measurement has been of interest to software engineers for almost as long as software has been developed. As the field of software engineering matures and greater understanding of software products and process is gained, the field of software measurement will also continue to progress.

THE END.

1. Appendix 1 - Entity-Relationship Model Set Definitions

In this Appendix 1, the sets that comprise the entity-relationship models of C++ class and object modularity described in Chapter 4, and the basic software model described in Chapter 6, are defined.

1.1. Entities

Class Entity *C*

The set C is the set of all classes. This is the equivalent of the entity CLASS in the entity relationship diagrams.

ci is a unique integer identifier used as the key field in relation C
name is the name of the class

C = {(ci, name) | (ci is a unique integer identifier) ∧ (name is a valid C++ class name) }

Method Entity *M* and Class Has Member Method MM

The set M is the set of all methods. This set is the equivalent of the entity METHOD in the entity relationship diagrams. A class may have member methods. This relationship is described by the set MM and is the equivalent of the Has Member relationship between the CLASS entity and the METHOD entity. Since this is a 1:N relationship, the set MM can be combined with the set M.

mi is a unique integer identifier used as the key field in relation M
ci is an identifier of an existing class
protection is the level of protection assigned to the method within the class
name is the name of the method
parameter_list is the text defining the method parameter list
purpose specifies the general purpose of the method
lines is a count of the number of lines of code within the method

M = {(mi, ci, protection, name, parameter_list, purpose, lines)} = {(mi, c.ci, protection, name, parameter_list, purpose, lines) | (mi is a unique integer identifier) ∧ c ∈ C ∧ (m is a member method of c) ∧ protection ∈ {public, protected, private} ∧ (name is a valid C++ method name) ∧ (parameter_list is the text defining the parameter list of the method) ∧ (purpose ∈ {constructor, destructor, operator, normal}) ∧ (lines is a count of the number of lines of code within the method)}

Attribute Entity *A* and Class Has Member Attribute *MA*

The set A is the set of all attributes. This set is the equivalent of the entity ATTRIBUTE in the entity relationship diagrams. A class may have member attributes. This relationship is described by the set MA and is the equivalent of the Has Member relationship between the CLASS entity and the ATTRIBUTE entity. Since this is a 1:N relationship, the set MA can be combined with the set A.

ai is a unique integer identifier used as the key field in relation A
ci is an identifier of an existing class
protection is the level of protection assigned to the attribute within the class
name is the name of the attribute
type specifies the type of the attribute

pointer specifies whether or not the attribute is a pointer
static specifies whether or not the attribute is declared to be static

A = {(ai, ci, protection, name, type, pointer, static)} = {(ai, c.ci, protection, name, type, pointer, static) | (ai is a unique integer identifier) ∧ c ∈ C ∧ (a is a member attribute of c) ∧ protection ∈ {public, protected, private} ∧ (name is a valid C++ attribute name) ∧ (type ∈ {char, short, int, long, unsigned char, signed char, unsigned short, signed short, unsigned int, signed int, unsigned long, signed long, float, double, long double}) ∧ (pointer ∈ {true, false}) ∧ (static ∈ {true, false})}

Object Entity *O* and Object Instance of Class *OIC*

The set O is the set of all objects. This is the equivalent of the entity OBJECT in the entity relationship diagrams. An object is an instance of a class. This relationship is the equivalent of the Instance Of *OIC* relationship between the CLASS entity and the OBJECT entity. Since OIC is a 1:N relationship, it can be combined with O to form one relation.

oi is a unique integer identifier used as the key field in relation O
ci is an identifier of the class from which the object is instantiated
name is the name of the object
pointer specifies whether or not the object is a pointer
global specifies whether or not the object is a global one

O = {(oi, ci, name, pointer, global)} = {(oi, c.ci, name, pointer, global) | (oi is a unique integer identifier) ∧ c ∈ C ∧ (name is a valid C++ object name) ∧ (pointer ∈ {true, false}) ∧ (global ∈ {true, false})}

Global Function Entity *F*

The set F is the set of all global functions. Attribute *fi* is a unique integer identifier used as the key field in relation F. Attribute *name* is the name of the global function. This is the equivalent of the entity GLOBAL FUNCTION in the entity relationship diagrams.

fi is a unique integer identifier used as the key field in relation F
name is the name of the global function

F = {(fi, name) | (fi is a unique integer identifier) ∧ (name is a valid C++ global function name)}

Global Variable Entity *V*

The set V is the set of all global variables. Attribute *vi* is a unique integer identifier used as the key field in relation V. Attribute *name* is the name of the global variable. This is the equivalent of the entity GLOBAL VARIABLE in the entity relationship diagrams.

vi is a unique integer identifier used as the key field in relation V
name is the name of the global variable
type specifies the type of the global variable

V = {(vi, name, type) | (vi is a unique integer identifier) ∧ (name is a valid C++ global variable name) ∧ (type ∈ {char, short, int, long, unsigned char, signed char, unsigned short, signed short, unsigned int, signed int, unsigned long, signed long, float, double, long double})}

1.2. Basic Relationships

Method Class-Writes Attribute *MCWriteA*

A method may directly class-write a value to an attribute. Class-writing describes the case where the method references the attribute using only its identifier name, or uses one or more class names, separated by colons, before the attribute name. This is the equivalent of the Class-Writes relationship between the METHOD and ATTRIBUTE entities.

mi is an identifier of an existing method
ai is an identifier of an existing attribute

MCWriteA = {(mi, ai)} = {(m.mi, a.ai) | m ∈ M ∧ a ∈ A ∧ (method m class-writes to attribute a)}

Method Class-Reads Attribute *MCReadA*

A method may directly class-read a value from an attribute. Class-reading describes the case where the method references the attribute using only its identifier name, or uses one or more class names, separated by colons, before the attribute name. This is the equivalent of the Class-Reads relationship between the METHOD and ATTRIBUTE entities.

mi is an identifier of an existing method
ai is an identifier of an existing attribute

MCReadA = {(mi, ai)} = {(m.mi, a.ai) | m ∈ M ∧ a ∈ A ∧ (method m class-reads from attribute a)}

Method Class-Invokes Method *MCInvM*

A method may directly class-invoke a method. Class-invoking describes the case where the method references the method using only its identifier name, or uses one or more class names, separated by colons, before the method name. This is the equivalent of the Class-Invokes relationship between METHOD entities

mi is an identifier of an existing method
invoked_mi is an identifier of an existing method

MCInvM = {(mi, invoked_mi)} = {(m.mi, x.mi) | m ∈ M ∧ x ∈ M ∧ m.mi ≠ x.mi ∧ (method mi directly class-invokes method invoked_mi)}

Class Inherits Parent Class *IP*

A class c may have immediate parent classes. This relationship is the equivalent of the Inherits Parent relationship between CLASS entities. The protection level assigned to the inherited parent is described by one of {public, protected, private}.

ci is an identifier of an existing class
parent_ci is an identifier of an existing class
protection is the level of protection assigned to the inheritance

IP = {(ci, parent_ci, protection)} = {(c.ci, x.ci, protection) | c ∈ C ∧ x ∈ C ∧ protection ∈ {public, protected, private} ∧ (x is an immediate parent class of class c with given inheritance protection level)}

Global Function Immediate Friend to Class *FF*

A class may have one or more friend global functions. This relationship is the equivalent of the Function Immediate Friend relationship between the CLASS entity and GLOBAL FUNCTION entity.

ci is an identifier of an existing class
friend_fi is an identifier of an existing global function

FF = {(ci, friend_fi)} = {(c.ci, f.fi) | c ∈ C ∧ f ∈ F ∧ (within the definition of class c, global function f is explicitly declared to be a friend)}

Global Function Has Immediate Object *FIMO*

A global function f may have immediate objects. These are objects that are instantiated within the body of the global function. This relationship is the equivalent of the Has Immediate relationship between the GLOBAL FUNCTION entity and the OBJECT entity.

fi is an identifier of an existing global function
oi is an identifier of an existing object

FIMO = {(fi, oi)} = {(f.fi, o.oi) | f ∈ F ∧ o ∈ O ∧ (o is an immediate object of f)}

Method Global Writes to Global Variable *MGWriteV*

A method may directly write a value to a global variable. This is the equivalent of the Global Writes relationship between the METHOD and GLOBAL VARIABLE entities.

mi is an identifier of an existing method
vi is an identifier of an existing global variable

MGWriteV = {(mi, vi)} = {(m.mi, v.vi) | m ∈ M ∧ v ∈ V ∧ (method m directly global-writes to global variable v)}

Method Global Reads From Global Variable *MGReadV*

A method may directly read a value from a global variable. This is the equivalent of the Global Reads relationship between the METHOD and GLOBAL VARIABLE entities.

mi is an identifier of an existing method
vi is an identifier of an existing global variable

MGReadV = {(mi, vi)} = {(m.mi, v.vi) | m ∈ M ∧ v ∈ V ∧ (method m directly global-reads from global variable v)}

Method Global Invokes Global Function *MGInvF*

A method may directly invoke a global function. This is the equivalent of the Global Invokes relationship between the METHOD and GLOBAL FUNCTION entities.

mi is an identifier of an existing method
fi is an identifier of an existing global function

MGInvF = {(mi, fi)} = {(m.mi, f.fi) | m ∈ M ∧ f ∈ F ∧ (method m directly global-invokes global function f)}

Class Has Member Object *MO*

A class c may have a member object. This relationship is the equivalent of the Has member relationship between the CLASS entity and the OBJECT entity.

ci is an identifier of an existing class
oi is an identifier of an existing object
protection is the level of protection of the object within the class ci

MO = {(ci, oi, protection)} = {(c.ci, o.oi, protection) | c ∈ C ∧ o ∈ O ∧ (o is a member object of c) ∧ protection ∈ {public, protected, private}}

Global Function Object-Accesses Object *FOAccessO*

A global function may access an object. This access may be reading or writing an object attribute or invoking an object method. This relationship is the equivalent of the Object Accesses relationship between the GLOBAL FUNCTION and OBJECT entities.

fi is an identifier of an existing global function
oi is an identifier of an existing object
action is the type of access from the global function to the object

FOAccessO = {(fi, oi, action)} = {(f.fi, o.oi, action) | f ∈ F ∧ o ∈ O ∧ action ∈ {read, write, invoke) ∧ (global function f directly object-accesses object o)}

Method Object-Accesses Object *MOAccessO*

A method may access an object. This access may be reading or writing an object attribute or invoking an object method. This relationship is the equivalent of the Object Accesses relationship between the METHOD and OBJECT entities.

mi is an identifier of an existing method
oi is an identifier of an existing object
action is the type of access from the method to the object

MOAccessO = {(mi, oi, action)} = {(m.mi, o.oi, action) | m ∈ M ∧ o ∈ O ∧ action ∈ {read, write, invoke) ∧ (method m directly object-accesses object o)}

Global Function Global Writes to Global Variable *FGWriteV*

A global function may directly write a value to a global variable. This is the equivalent of the Global Writes relationship between the GLOBAL FUNCTION and GLOBAL VARIABLE entities.

fi is an identifier of an existing global function
vi is an identifier of an existing global variable

FGWriteV = {(fi, vi)} = {(f.fi, v.vi) | f ∈ F ∧ v ∈ V ∧ (global function f directly global-writes to global variable v)}

Method Immediate Friend to Class *FM*

A class may have one or more friend methods. This relationship is the equivalent of the Method Immediate Friend relationship between the CLASS entity and METHOD entity in the Basic Software Document Model. This set does not appear in the Software Measurement Model.

ci is an identifier of an existing class
friend_mi is an identifier of an existing method

FM = {(ci, friend_mi)} = {(c.ci, m.mi) | c ∈ C ∧ m ∈ M ∧ (within the definition of class c, method m is explicitly declared to be a friend)}

Class Immediate Friend to Class *FC*

A class may have one or more friend classes. This relationship is the equivalent of the Class Immediate Friend relationship between the CLASS entity and METHOD entity in the Basic Software Document Model. This set does not appear in the Software Measurement Model.

ci is an identifier of an existing class

friend_ci is an identifier of an existing class

FC = {(ci, friend_ci)} = {(c.ci, x.ci) | c ∈ C ∧ x ∈ C ∧ (within the definition of class c, class x is explicitly declared to be a friend)}

Global Function Within Scope of Class *SF*

A class may have one or more global functions within its scope. This relationship is the equivalent of the Function Within Scope relationship between the CLASS and GLOBAL FUNCTION entities.

ci is an identifier of an existing class
fi is an identifier of an existing global function

SF = {(ci, fi)} = {(c.ci, f.fi) | c ∈ C ∧ f ∈ F ∧ (the global function f is within the scope of class c such that class c is potentially able to invoke the global function)}

Global Variable Within Scope of Class *SV*

A class may have one or more global variables within its scope. This relationship is the equivalent of the Variable Within Scope relationship between the CLASS and GLOBAL VARIABLE entities.

ci is an identifier of an existing class
vi is an identifier of an existing global variable

SV = {(ci, vi)} = {(c.ci, v.vi) | c ∈ C ∧ v ∈ V ∧ (the global variable v is within the scope of class c such that class c is potentially able to access the global variable)}

Global Object Within Scope of Class *SO*

A class may have one or more global objects within its scope. This relationship is the equivalent of the Global Object Within Scope relationship between the CLASS and OBJECT entities.

ci is an identifier of an existing class
oi is an identifier of an existing object

SO = {(ci, oi)} = {(c.ci, o.oi) | c ∈ C ∧ o ∈ O ∧ (the global object o is within the scope of class c such that class c is potentially able to access the global object)}

1.3. Derived Relationships

Class Has Accessible Attribute *AA*

A class c may have accessible inherited attributes. This relationship is the equivalent of the Has Accessible relationship between the CLASS entity and the ATTRIBUTE entity.

ci is an identifier of an existing class
ai is an identifier of an existing attribute
protection is the level of protection of the attribute within the class ci

$AA = \{(ci, ai, protection)\} = \{(c.ci, a.ai, protection) \mid c \in C \wedge a \in A \wedge protection \in \{public, protected, private\} \wedge (a \text{ is an inherited attribute of c with given protection level})\}$

Class Has Inaccessible Attribute *IAA*

A class c may have inaccessible inherited attributes. This relationship is the equivalent of the Has Inaccessible relationship between the CLASS entity and the ATTRIBUTE entity.

ci is an identifier of an existing class
ai is an identifier of an existing attribute
protection = inaccessible

$IAA = \{(ci, ai, protection)\} = \{(c.ci, a.ai, protection) \mid c \in C \wedge a \in A \wedge ((a \text{ is a private member or inherited attribute of an immediate parent of class c}) \vee (a \text{ is an inaccessible attribute of an immediate parent of class c})) \wedge protection = inaccessible\}$

Class Has Object Attribute *OA*

To simplify many of the object modularity measure definitions, the set OA of object attributes is defined. This set contains all the member and inherited attributes of each class.

ci is an identifier of an existing class
ai is an identifier of an existing attribute
protection ∈ {public, protected, private, inaccessible}

$OA = \{(ci, ai, protection)\} = AA \cup IAA \cup \{(x.ci, x.ai, x.protection) \mid x \in A\}$

Class Has Accessible Method *AM*

A class c may have accessible inherited methods. This relationship is the equivalent of the Has Accessible relationship between the CLASS entity and the METHOD entity. The *purpose* attribute is carried over unchanged from the original purpose of the attribute in its member class. Although this represents a redundancy in the database, it greatly simplifies some measure definitions.

ci is an identifier of an existing class
mi is an identifier of an existing method
protection is the level of protection of the method within the class ci
purpose specifies the general purpose of the method

$AM = \{(ci, mi, protection, purpose)\} = \{(c.ci, m.mi, protection) \mid c \in C \wedge m \in M \wedge protection \in \{public, protected, private\} \wedge (purpose \in \{constructor, destructor, operator, normal\}) \wedge (m \text{ is an inherited method of c with given protection level})\}$

Class Has Inaccessible Method *IAM*

A class c may have inaccessible inherited methods. This relationship is the equivalent of the Has Inaccessible relationship between the CLASS entity and the METHOD entity. The *purpose* attribute is carried over unchanged from the original purpose of the attribute in its member class. Although this represents a redundancy in the database, it greatly simplifies some measure definitions.

ci is an identifier of an existing class
mi is an identifier of an existing method
protection = inaccessible
purpose specifies the general purpose of the method

IAM = {(ci, mi, protection, purpose)} = {(c.ci, m.mi) | c ∈ C ∧ m ∈ M ∧ ((m is a private member or inherited method of an immediate parent of class c) ∨ (m is an inaccessible method of an immediate parent of class c)) ∧ protection = inaccessible ∧ (purpose ∈ {constructor, destructor, operator, normal})}

Class Has Object Method *OM*

To simplify many of the object modularity measure definitions, the set OM of object methods is defined. This set contains all the member and inherited methods of each class.

ci is an identifier of an existing class
mi is an identifier of an existing method
protection ∈ {public, protected, private, inaccessible}

OM = {(ci, mi, protection)} = AM ∪ IAM ∪ {(x.ci, x.mi, x.protection) | x ∈ M}

Class Has Immediate Object *IMO*

A class c may have immediate objects. These are objects that are member, accessible or inaccessible objects of the class. This relationship is the equivalent of the Has Immediate relationship between the CLASS entity and the OBJECT entity.

ci is an identifier of an existing class
oi is an identifier of an existing object
protection is the level of protection of the object within the class.

IMO = {(ci, oi, protection)} = {(c.ci, o.oi, protection) | c ∈ C ∧ o ∈ O ∧ (o is an immediate object of c) ∧ protection ∈ {public, protected, private, inaccessible}}

Method Indirectly Same Class-Writes Attribute *MICWriteA*

A method may indirectly class-write a value to an attribute that is a member of the same class. This occurs when the method invokes a same class method that itself directly or indirectly same class-writes an attribute. This is the equivalent of the Indirectly Same Class-Writes relationship between the METHOD and ATTRIBUTE entities.

mi is an identifier of an existing method
ai is an identifier of an existing attribute

MICWriteA = {(mi, ai)} = {(m.mi, a.ai) | m ∈ M ∧ a ∈ A ∧ m.ci = a.ci ∧ (method m indirectly class-writes to attribute a)}

Method Indirectly Same Class-Reads Attribute *MICReadA*

A method may indirectly class-read a value to an attribute that is a member of the same class. This occurs when the method invokes a same class method that itself directly or indirectly same class-reads an attribute. This is the equivalent of the Indirectly Same Class-Reads relationship between the METHOD and ATTRIBUTE entities.

mi is an identifier of an existing method
ai is an identifier of an existing attribute

MICReadA = {(mi, ai)} = {(m.mi, a.ai) | m ∈ M ∧ a ∈ A ∧ m.ci = a.ci ∧ (method m indirectly class-reads from attribute a)}

Method Indirectly Same Class-Invokes Method *MICInvM*

A method may indirectly class-invoke a method. This occurs when the method invokes a same class method that itself directly or indirectly same class-invokes a method. This is the equivalent of the Indirectly Same Class-Invokes relationship between METHOD entities

mi is an identifier of an existing method
invoked_mi is an identifier of an existing method

MICInvM = {(mi, invoked_mi)} = {(m.mi, x.mi) | m ∈ M ∧ x ∈ M ∧ m.mi ≠ x.mi ∧ m.ci = x.ci ∧ (method m indirectly class-invokes method invoked_mi)}

Method Indirectly Same Object Class-Writes Attribute *MIOCWriteA*

A method may indirectly class-write a value to an attribute that is a member of the same object. This occurs when the method invokes a same object method that itself directly or indirectly same object class-writes an attribute. This is the equivalent of the Indirectly Same Object Class-Writes relationship between the METHOD and ATTRIBUTE entities.

mi is an identifier of an existing method
ai is an identifier of an existing attribute

MIOCWriteA = {(mi, ai)} = {(m.mi, a.ai) | m ∈ OM ∧ a ∈ OA ∧ m.ci = a.ci ∧ (method m indirectly class-writes to attribute a)}

Method Indirectly Same Object Class-Reads Attribute *MIOCReadA*

A method may indirectly class-read a value to an attribute that is a member of the same object. This occurs when the method invokes a same object method that itself directly or indirectly same object class-reads an attribute. This is the equivalent of the Indirectly Same Object Class-Reads relationship between the METHOD and ATTRIBUTE entities.

mi is an identifier of an existing method
ai is an identifier of an existing attribute

MIOCReadA = {(mi, ai)} = {(m.mi, a.ai) | m ∈ OM ∧ a ∈ OA ∧ m.ci = a.ci ∧ (method m indirectly class-reads from attribute a)}

Method Indirectly Same Object Class-Invokes Method *MIOCInvM*

A method may indirectly class-invoke a method that is a member of the same object. This occurs when the method invokes a same object method that itself directly or indirectly same object class-invokes a method. This is the equivalent of the Indirectly Same Object Class-Invokes relationship between METHOD entities

mi is an identifier of an existing method
invoked_mi is an identifier of an existing method

MIOCInvM = {(mi, invoked_mi)} = {(m.mi, x.mi) | m ∈ OM ∧ x ∈ OM ∧ m.mi ≠ x.mi ∧ m.ci = x.ci ∧ (method m indirectly class-invokes method invoked_mi)}

Class Inherits Distant Ancestor Class *IDA*

A class c may have distant ancestor classes. This relationship is the equivalent of the Inherits Distant Ancestor relationship between CLASS entities.

ci is an identifier of an existing class
ancestor_ci is an identifier of an existing class

IDA = {(ci, ancestor_ci)} = {(c.ci, x.ci) | c ∈ C ∧ x ∈ C ∧ (x is a distant ancestor class of class c)}

Class Element Immediate Friend to Class *CEF*

A class may have one or more friend classes that have member elements with a friend relationship to the class. This relationship is the equivalent of the Class Element Immediate Friend relationship between CLASS entities.

ci is an identifier of an existing class
friend_ci is an identifier of an existing class

CEF = {(ci, friend_ci)} = {(c.ci, x.ci) | c ∈ C ∧ x ∈ C ∧ (class x or member elements of class x are immediate friends to class c)}

Class Element Inherited Friend to Class *CIF*

A class may have inherited ancestor classes that have one or more friend classes that have member elements with a friend relationship to the ancestor class. This relationship is the equivalent of the Class Element Inherited Friend relationship between CLASS entities.

ci is an identifier of an existing class
friend_ci is an identifier of an existing class

CIF = {(ci, friend_ci)} = {(c.ci, x.ci) | c ∈ C ∧ x ∈ C ∧ (class x or member elements of class x are immediate friends to an ancestor of class c)}

Global Function Inherited Friend to Class *FIF*

A class may have inherited ancestor classes that have one or more friend global functions. This relationship is the equivalent of the Function Inherited Friend relationship between the CLASS entity and GLOBAL FUNCTION entity.

ci is an identifier of an existing class
friend_fi is an identifier of an existing global function

FIF = {(ci, friend_fi)} = {(c.ci, f.fi) | c ∈ C ∧ f ∈ F ∧ (global function f is an immediate friend to an ancestor of class c)}

Class Has Accessible Object *AO*

A class c may have an inherited accessible object. This relationship is the equivalent of the Has Accessible relationship between the CLASS entity and the OBJECT entity.

ci is an identifier of an existing class
oi is an identifier of an existing object
protection is the level of protection of the object within the class ci

AO = {(ci, oi, protection)} = {(c.ci, o.oi, protection) | c ∈ C ∧ o ∈ O ∧ protection ∈ {public, protected, private} ∧ (o is an inherited object of c with given protection level)}

Class Has Inaccessible Object *IAO*

A class c may have inherited inaccessible objects. This relationship is the equivalent of the Has Inaccessible relationship between the CLASS entity and the OBJECT entity.

ci is an identifier of an existing class
oi is an identifier of an existing object
protection = inaccessible

IAO = {(ci, oi, protection)} = {(c.ci, o.oi, protection) | c ∈ C ∧ o ∈ O ∧ protection = inaccessible ∧ ((o is a private member or private accessible or inaccessible object of an immediate parent of class c)) }

2. Appendix 2 - Basic Model to Measurement Model Transformations

2.1. Transformation 1 - AA and IAA

The set AA of accessible inherited attributes of a class is defined as follows.

AA = {(ci, ai, protection)} = {(x.ci, a.ai, protection) | x ∈ IP ∧ a ∈ A ∧ (attribute a is an inherited attribute of class x.ci ∧ protection ∈ {public, protected, private})}

The set IAA of inaccessible inherited attributes of a class is defined as follows.

IAA = {(ci, ai, protection)} = {(x.ci, a.ai, inaccessible) | x ∈ IP ∧ a ∈ A ∧ (attribute a is an inherited but inaccessible attribute of class x.ci)}

An attribute inherited from a parent may be a member attribute of the parent or may be itself an inherited attribute of the parent class. An attribute may be inherited more than once by a class. This is because it may be inherited through different paths in the inheritance hierarchy. If this is the case, only a single instance of the attribute will be recorded. Any measures made from the inherited attribute data must not distinguish between multiple inheritance of a single attribute.

As attributes are inherited, their level of protection can change. The level of protection assigned depends on the level of protection the attribute has within the immediate parent class, and the level of protection assigned to the parent inheritance.

Constructing the sets of accessible and inaccessible attributes for each class from the collected data requires the recursive application of an algorithm. Ross and Wright (1992, p. 188), define a recursive set by a starting point (B) and by a relation (R) that is repeatedly applied using the previous values. Similarly, the algorithm that must be repeatedly applied here to produce the set of accessible inherited attributes has a starting point we shall call (B) and a repeated element (R).

To determine the sets AA and IAA, the starting point is derived from the set of basic parents BP of classes that are parent classes but themselves have no parents.

BP = {(ci, parent_ci, protection)} = {(x.ci, x.parent_ci, x.protection) | x ∈ IP ∧ ¬∃y{y | y ∈ IP ∧ y.ci = x.parent_ci}}

Starting Point (B)

$Start\_AA_{public}$ = {(ci, ai, protection)} = {(x.ci, y.ai, public) | x ∈ BP ∧ y ∈ A ∧ x.parent_ci = y.ci ∧ x.protection = y.protection = public }

$Start\_AA_{protected1}$ = {(ci, ai, protection)} = {(x.ci, y.ai, protected) | x ∈ BP ∧ y ∈ A ∧ x.parent_ci = y.ci ∧ x.protection = public ∧ (y.protection = protected)}

$Start\_AA_{protected2}$ = {(ci, ai, protection)} = {(x.ci, y.ai, protected)| x ∈ BP ∧ y ∈ A ∧ x.parent_ci = y.ci ∧ x.protection = protected ∧ y.protection ∈ {public, protected}}

$Start\_AA_{private}$ = {(ci, ai, protection)} = {(x.ci, y.ai, private) | x ∈ BP ∧ y ∈ A ∧ x.parent_ci = y.ci ∧ x.protection = private ∧ y.protection ∈ {public, protected}}

$AA = Start\_AA_{public} \cup Start\_AA_{protected1} \cup Start\_AA_{protected2} \cup Start\_AA_{private}$

$IAA = \{(ci, ai, protection)\} = \{(x.ci, y.ai, inaccessible) \mid x \in BP \wedge y \in A \wedge x.parent\_ci = y.ci \wedge y.protection = private \}$

$New\_AA = AA \cup IAA$

The set NP is the set of descendants of the base parents that are themselves parents to other classes.

$NP = \{(ci, parent\_ci, protection)\} = \{(x.ci, x.parent\_ci, x.protection) \mid x \in IP \wedge \exists y\{y \mid y \in BP \wedge y.ci = x.parent\_ci\}\}$

Repeated Algorithm (R)
Once the start sets have been constructed, the repeated application of the algorithm (R) constructs the full sets AA and IAA. In (R), class parent_ci itself has parent classes. Algorithm (R) is applied until all parent child relationships (ci, parent_ci) ∈ IP have been investigated for possible attribute inheritance.

repeat {

$Repeat\_AA_{public1} = \{(ci, ai, protection)\} = \{(x.ci, y.ai, public) \mid x \in NP \wedge y \in A \wedge y.ci = x.parent\_ci \wedge x.protection = public \wedge y.protection = public\}$

$Repeat\_AA_{public2} = \{(ci, ai, protection)\} = \{(x.ci, y.ai, public) \mid x \in NP \wedge y \in New\_AA \wedge x.parent\_ci = y.ci \wedge x.protection = public \wedge (y.protection = public)\}$

$Repeat\_AA_{protected1} = \{(ci, ai, protection)\} = \{(x.ci, y.ai, protected) \mid x \in NP \wedge y \in A \wedge x.parent\_ci = y.ci \wedge x.protection = public \wedge y.protection = protected\}$

$Repeat\_AA_{protected2} = \{(ci, ai, protection)\} = \{(x.ci, y.ai, protected) \mid x \in NP \wedge y \in New\_AA \wedge x.parent\_ci = y.ci \wedge x.protection = public \wedge y.protection = protected\}$

$Repeat\_AA_{protected3} = \{(ci, ai, protection)\} = \{(x.ci, y.ai, protected) \mid x \in NP \wedge y \in A \wedge x.parent\_ci = y.ci \wedge x.protection = protected \wedge y.protection \in \{public, protected\}\}$

$Repeat\_AA_{protected4} = \{(ci, ai, protection)\} = \{(x.ci, y.ai, protected) \mid x \in NP \wedge y \in New\_AA \wedge x.parent\_ci = y.ci \wedge x.protection = protected \wedge y.protection \in \{public, protected\}\}$

$Repeat\_AA_{private1} = \{(ci, ai, protection)\} = \{(x.ci, y.ai, private) \mid x \in NP \wedge y \in A \wedge x.parent\_ci = y.ci \wedge x.protection = private \wedge y.protection \in \{public, protected\}\}$

$Repeat\_AA_{private2} = \{(ci, ai, protection)\} = \{(x.ci, y.ai, private) \mid x \in NP \wedge y \in New\_AA \wedge x.parent\_ci = y.ci \wedge x.protection = private \wedge y.protection \in \{public, protected\}\}$

$Repeat\_AA = Repeat\_AA_{public1} \cup Repeat\_AA_{public2} \cup Repeat\_AA_{protected1} \cup Repeat\_AA_{protected2} \cup Repeat\_AA_{protected3} \cup Repeat\_AA_{protected4} \cup Repeat\_AA_{private1} \cup Repeat\_AA_{private2}$

$AA = AA \cup Repeat\_AA$

$Repeat\_IAA_1 = \{(ci, ai, protection)\} = \{(x.ci, y.ai, inaccessible) \mid x \in NP \wedge y \in A \wedge x.parent\_ci = y.ci \wedge y.protection = private\}$

$Repeat\_IAA_2 = \{(ci, ai, protection)\} = \{(x.ci, y.ai, inaccessible) \mid x \in NP \wedge y \in New\_AA \wedge x.parent\_ci = y.ci \wedge y.protection \in \{private, inaccessible\}\}$

$Repeat\_IAA = Repeat\_IAA_1 \cup Repeat\_IAA_2$

$IAA = IAA \cup Repeat\_IAA$

$New\_AA = Repeat\_AA \cup Repeat\_IAA$

$New\_NP = \{(ci, parent\_ci, protection)\} = \{(x.ci, x.parent\_ci, x.protection) \mid x \in IP \wedge \exists y\{y \mid y \in NP \wedge y.ci = x.parent\_ci\}\}$

$NP = New\_NP$

} until ($New\_NP = \varnothing$)

For classes in the system that have inherited attributes, the resultant set AA is a set of the inherited accessible attributes and IAA is a set of the inherited inaccessible attributes.

2.2. Transformation 2 - AM and IAM

The set AM of accessible inherited methods of a class is defined as follows.

$AM = \{(ci, mi, protection)\} = \{(x.ci, m.mi, protection) \mid x \in IP \wedge m \in M \wedge$ (method m is an inherited method of class x.ci $\wedge$ protection $\in$ {public, protected, private})}

The set IAM of inaccessible inherited methods of a class is defined as follows.

$IAM = \{(ci, mi, protection)\} = \{(x.ci, m.mi, inaccessible) \mid x \in IP \wedge m \in M \wedge$ (method m is an inherited but inaccessible method of class x.ci)}

The sets AM and IAM are derived in the same way as sets AA and IAA. The algorithm given previously for the derivation of sets AA and IAA is used with the substitution of set M for set A, set AM for set AA and set IAM for set IAA.

2.3. Transformation 3 - *AO* and *IAO*

The set AO of accessible inherited objects of a class is defined as follows.

$AO = \{(ci, oi, protection)\} = \{(x.ci, o.oi, protection) \mid x \in IP \wedge o \in MO \wedge$ (object o is an inherited object of class x.ci $\wedge$ protection $\in$ {public, protected, private})}

The set IAO of inaccessible inherited objects of a class is defined as follows.

IAO = {(ci, oi, protection)} = {(x.ci, o.oi, inaccessible) | x ∈ IP ∧ o ∈ MO ∧ (object o is an inherited but inaccessible object of class x.ci)}

The sets AO and IAO are derived in the same way as sets AA and IAA. The algorithm given previously for the derivation of sets AA and IAA is used with the substitution of set O for set A, set AO for set AA and set IAO for set IAA.

2.4. Transformation 4 - *IMO*

The set IMO of immediate objects of a class is defined as follows.

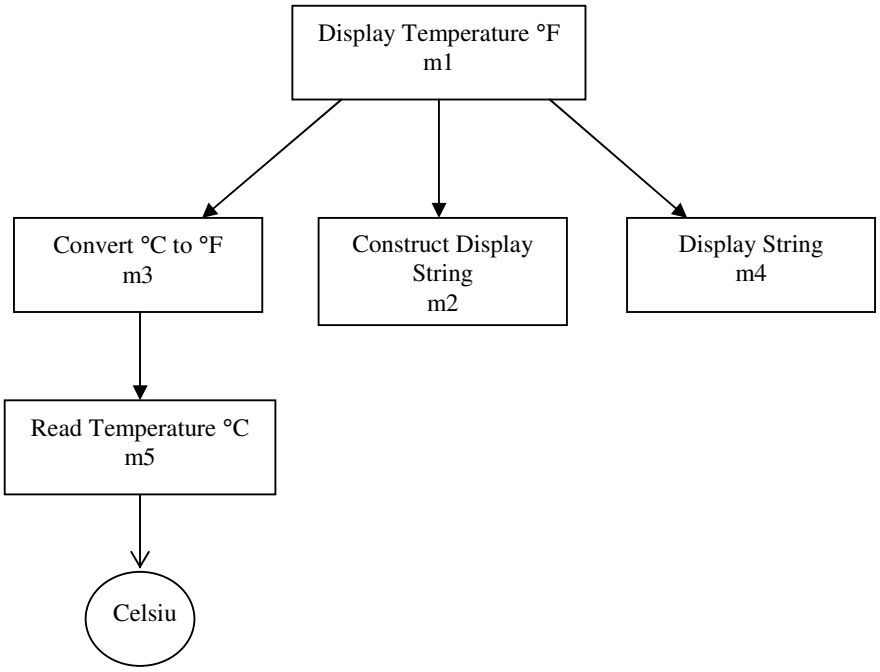
IMO = {(ci, oi, protection)} = {(x.ci, o.oi, protection) | x ∈ IP ∧ o ∈ MO ∧ (object MO is a member or inherited object of class x.ci ∧ protection ∈ {public, protected, private, inaccessible})}

IMO = MO ∪ AO ∪ IAO

2.5. Transformation 5 - *MICReadA*

A class member method indirectly class-reads a same class member attribute if it class-invokes a same class member method that itself either directly or indirectly reads the same class member attribute. To determine the set of attributes indirectly read by a method, a path matrix must be constructed showing all the possible invocation paths between same class methods. If a path exists, the path matrix will have a 1 in the position corresponding to a connection between two same class methods. If no path exits, the path matrix value for that connection will be 0.

For example, the following diagram shows a simple calling structure between methods m1 to m5 and attribute Celsius.



Method m5 directly reads from attribute Celsius and methods m3 and m1 indirectly read from attribute Celsius. Methods m2 and m4 neither directly nor indirectly read from attribute Celsius.

The path matrix associated with this the above example is given below.

		j				
		m1	**m2**	**m3**	**m4**	**m5**
i	**m1**	0	1	1	1	1
	m2	0	0	0	0	0
	m3	0	0	0	0	1
	m4	0	0	0	0	0
	m5	0	0	0	0	0

To determine the attributes indirectly accessed by each method, the following algorithm must be applied.

Use the sets method same class-reads attribute MSCReadA and method same class invokes method MSCInvM.

$$MSCReadA = \{(mi, ai)\} = \{(x.mi, y.ai) \mid x \in M \wedge y \in A \wedge (x.ci = y.ci) \wedge (x.mi, y.ai) \in MCReadA\}$$

$$MSCInvM = \{(mi, invoked\_mi)\} = \{(x.mi, y.mi) \mid x \in M \wedge y \in M \wedge (x.ci = y.ci) \wedge (x.mi, y.mi) \in MCInvM\}$$

Grassmann and Tremblay (1996, p. 363) define a directed graph G = (V, E) where V is the set of vertices and E the set of edges of the graph. For a directed graph, the edges form ordered pairs $\langle v_i, v_j \rangle$ where v_i = the vertex from where the edge begins, and v_j the vertex where the edge ends.

$$MCInvMGraph(V, E) \quad \text{where the vertices } V = \{v : v \in M\}$$
$$\text{and edges } E = \{\langle v_i, v_j \rangle : \langle v_i, v_j \rangle \in MSCInvM\}$$

Grassmann and Tremblay (1996, p. 378) then, from graph MCInvMGraph(V, E), form an $n \times n$ adjacency matrix MCInvMAdjacency, where n = #M, i identifies the row, and j the column of the graph, and whose elements a_{ij} are given by

$a_{ij} = 1$, if $\langle v_i, v_j \rangle \in E$

$a_{ij} = 0$, otherwise.

Using Warshall's algorithm (Grassmann & Tremblay 1996, pp. 383-385) calculate the path matrix MCInvMPath from the adjacency matrix MCInvMAdjacency. Each element of the path matrix,

$p_{ij} = 1$ if an invocation path exists from v_i to v_j

$p_{ij} = 0$ if no invocation path exists from v_i to v_j

Another way of writing p_{ij} is MCInvMPath$\langle m_i, m_j \rangle$ where m_i is the method from which the invocation originates and m_j is the method invoked. If $p_{ij} = 1$ and $i \neq j$ then member method m_i indirectly reads the attributes that method m_j directly reads.

The set MICReadA of class member attributes indirectly read from by same class member methods is defined as follows.

MICReadA = {(mi, ai)} = {(x.mi, y.ai) | x ∈ M ∧ y ∈ MSCReadA ∧ (MCInvMPath⟨x.mi, y.mi⟩ = 1) }

For the example given above, MICReadA = {(Convert °C to °F, Celsius), (Display Temperature °F, Celsius)}

2.6. Transformation 6 - *MICWriteA*

A class member method indirectly writes a same class member attribute if it invokes a same class method that itself either directly or indirectly writes that attribute. The same algorithm as that used to determine the set of methods indirectly reading from an attribute is applied here. Use the sets MSCWriteA and MSCInvM and construct the directed graph MCInvMGraph(V, E).

MSCWriteA = {(mi, ai)} = {(x.mi, y.ai) | x ∈ M ∧ y ∈ A ∧ (x.ci = y.ci) ∧ (x.mi, y.ai) ∈ MCWriteA}

MSCInvM = {(mi, invoked_mi)} = {(x.mi, y.mi) | x ∈ M ∧ y ∈ M ∧ (x.ci = y.ci) ∧ (x.mi, y.mi) ∈ MCInvM}

MCInvMGraph(V, E) where the vertices V = {v : v ∈ M}
and edges E = {$\langle v_i, v_j \rangle$: $\langle v_i, v_j \rangle$ ∈ MSCInvM}

From this, as before determine the adjacency matrix MCInvMAdjacency and then the path matrix MCInvMPath.

The set MICWriteA of attributes indirectly written to by same class member methods is defined as follows.

MICWriteA = {(mi, ai)} = {(x.mi, y.ai) | x ∈ M ∧ y ∈ MSCWriteA ∧ (MCInvMPath⟨x.mi, y.mi⟩ = 1)}

2.7. Transformation 7 - *MICInvM*

A class member method indirectly invokes a same class member method if it invokes a same class method that itself either directly or indirectly invokes that method. The same algorithm as that used to determine the set of methods indirectly reading from an attribute is applied here. Use the set MSCInvM and construct the directed graph MCInvMGraph(V, E).

MSCInvM = {(mi, invoked_mi)} = {(x.mi, y.mi) | x ∈ M ∧ y ∈ M ∧ (x.ci = y.ci) ∧ (x.mi, y.mi) ∈ MCInvM}

MCInvMGraph(V, E) where the vertices $V = \{v : v \in M\}$
and edges $E = \{\langle v_i, v_j\rangle : \langle v_i, v_j\rangle \in MSCInvM\}$

From this, as before determine the adjacency matrix MCInvMAdjacency and then the path matrix MCInvMPath.

The set MICInvM of methods indirectly invoked by same class member methods is defined as follows.

$MICInvM = \{(mi, invoked\_mi)\} = \{(x.mi, y.invoked\_mi) \mid x \in M \wedge y \in MSCInvM \wedge (MCInvMPath\langle x.mi, y.invoked\_mi\rangle = 1)\}$

2.8. Transformation 8 - *MIOCReadA*

A class member method indirectly reads a same object attribute if it invokes a same object method that itself either directly or indirectly reads that attribute. Use the sets OM, OA, MSOCReadA and MSOCInvM and construct the directed graph MOCInvMGraph(V, E).

$OM = \{(ci, mi, protection)\} = \{(x.ci, x.mi, x.protection) \mid x \in M\} \cup \{(x.ci, x.mi) \mid x \in AM\} \cup \{(x.ci, x.mi) \mid x \in IAM\}$

$OA = \{(ci, ai, protection)\} = \{(x.ci, x.ai, x.protection) \mid x \in A\} \cup \{(x.ci, x.ai) \mid x \in AA\}$

$MSOCReadA = \{(mi, ai)\} = \{(x.mi, y,ai) \mid x \in OM \wedge y \in OA \wedge x.ci = y.ci\ A\ (x.mi, y.ai) \in MCReadA\}$

$MSOCInvM = \{(mi, invoked\_mi)\} = \{(x.mi, y.mi) \mid x \in OM \wedge y \in OM \wedge x.ci = y.ci \wedge (x.mi, y.mi) \in MCInvM\}$

MOCInvMGraph(V, E) where the vertices $V = \{v : v \in OM\}$
and edges $E = \{\langle v_i, v_j\rangle : \langle v_i, v_j\rangle \in MSOCInvM\}$

From this, as before determine the adjacency matrix MOCInvMAdjacency and then the path matrix MOCInvMPath.

The set MIOCReadA of attributes indirectly written to by same object methods is defined as follows.

$MIOCReadA = \{(mi, ai)\} = \{(x.mi, y.ai) \mid x \in OM \wedge y \in MSOCReadA \wedge (MOCInvMPath\langle x.mi, y.mi\rangle = 1)\}$

2.9. Transformation 9 - *MIOCWriteA*

A class member method indirectly writes a same object attribute if it invokes a same object method that itself either directly or indirectly writes that attribute. Use the sets OM, OA, MSOCWriteA and MSOCInvM and construct the directed graph MOCInvMGraph(V, E).

$OM = \{(ci, mi, protection)\} = \{(x.ci, x.mi, x.protection) \mid x \in M\} \cup \{(x.ci, x.mi) \mid x \in AM\} \cup \{(x.ci, x.mi) \mid x \in IAM\}$

$OA = \{(ci, ai, protection)\} = \{(x.ci, x.ai, x.protection) \mid x \in A\} \cup \{(x.ci, x.ai) \mid x \in AA\}$

$MSOCWriteA = \{(mi, ai)\} = \{(x.mi, y,ai) \mid x \in OM \wedge y \in OA \wedge x.ci = y.ci\ A\ (x.mi, y.ai) \in MCWriteA\}$

$MSOCInvM = \{(mi, invoked\_mi)\} = \{(x.mi, y.mi) \mid x \in OM \wedge y \in OM \wedge x.ci = y.ci \wedge (x.mi, y.mi) \in MCInvM\}$

MOCInvMGraph(V, E) where the vertices $V = \{v : v \in OM\}$
and edges $E = \{\langle v_i, v_j\rangle : \langle v_i, v_j\rangle \in MSOCInvM\}$

From this, as before determine the adjacency matrix MOCInvMAdjacency and then the path matrix MOCInvMPath.

The set MIOCWriteA of attributes indirectly written to by same object methods is defined as follows.

$MIOCWriteA = \{(mi, ai)\} = \{(x.mi, y.ai) \mid x \in OM \wedge y \in MSOCWriteA \wedge (MOCInvMPath\langle x.mi, y.mi\rangle = 1)\}$

2.10. Transformation 10 - *MIOCInvM*

A class member method indirectly invokes a same object method if it invokes a same object method that itself either directly or indirectly invokes that method. Use the sets OM and MSOCInvM and construct the directed graph MOCInvMGraph(V, E).

$OM = \{(ci, mi, protection)\} = \{(x.ci, x.mi, x.protection) \mid x \in M\} \cup \{(x.ci, x.mi) \mid x \in AM\} \cup \{(x.ci, x.mi) \mid x \in IAM\}$

$MSOCInvM = \{(mi, invoked\_mi)\} = \{(x.mi, y.mi) \mid x \in OM \wedge y \in OM \wedge x.ci = y.ci \wedge (x.mi, y.mi) \in MCInvM\}$

MOCInvMGraph(V, E) where the vertices $V = \{v : v \in OM\}$
and edges $E = \{\langle v_i, v_j\rangle : \langle v_i, v_j\rangle \in MSOCInvM\}$

From this, as before determine the adjacency matrix MOCInvMAdjacency and then the path matrix MOCInvMPath.

The set MIOCInvM of methods indirectly invoked by same object methods is defined as follows.

$MIOCInvM = \{(mi, mi)\} = \{(x.mi, y.invoked\_mi) \mid x \in OM \wedge y \in MSOCInvM \wedge (MOCInvMPath\langle x.mi, y.invoked\_mi\rangle = 1)\}$

2.11. Transformation 11 - *CEF*

The friendship relationship between classes can be defined on a class to class level or on a class method to class level.

The set CEF of classes with their friend classes is defined as follows.

$$CEF = \{(ci, friend\_ci)\} = FC \cup \{(x.ci, y.ci) \mid x \in FM \wedge y \in M \wedge (x.mi = y.mi)\}$$

2.12. Transformation 12 - *IDA*

To determine the set IDA, an intermediate set class ancestor classes CAS must be constructed.

$CAS = \{(ci, ancestor\_ci)\} = \{(x.ci, y.ci) \mid x \in C \wedge y \in C \wedge$ y is an immediate or distant ancestor of class x$\}$

Constructing the set CAS requires the recursive application of an algorithm. To determine the set CAS, the starting point is derived from the set of class, distant ancestor classes where there is only one intervening class.

$DIS = \{(ci, ancestor\_ci)\} = \{(x.ci, y.parent\_ci) \mid x \in IP \wedge y \in IP \wedge x.parent\_ci = y.ci\}$

Starting Point (B)

$Start\_CAS = DIS$

$CAS = Start\_CAS$
$New\_CAS = CAS$

Repeated Algorithm (R)
Once the start sets have been constructed, the repeated application of the algorithm (R) constructs the full set CAS. In (R), class ancestor_ci itself has parent classes. Algorithm (R) is applied until all parent child relationships $(ci, parent\_ci) \in IP$ have been investigated.

repeat {

$Repeat\_CAS = \{(ci, ancestor\_ci)\} = \{(x.ci, y.ancestor\_ci) \mid x \in IP \wedge y \in New\_CAS \wedge x.parent\_ci = y.ci\} \cup \{(x.ci, y.ancestor\_ci) \mid x \in IP \wedge y \in CAS \wedge x.parent\_ci = y.ci\}$

$CAS = CAS \cup Repeat\_CAS$
$New\_CAS = Repeat\_CAS$

} until ($New\_CAS = \varnothing$)

For classes in the system that have parents, the resultant set CAS is a set of immediate and distant ancestor classes.

The set IDA of distant ancestor classes is defined as follows:

$IDA = \{(ci, ancestor\_ci)\} = \{(x.ci, x.ancestor\_ci) \mid x \in CAS \wedge \neg\exists y\{y \mid y \in IP \wedge y.ci = x.ci \wedge y.parent\_ci = x.ancestor\_ci\}\}$

2.13. Transformation 13 - *CIF*

If a child class inherits from ancestor classes that have friends defined then the child class has a partial friend relationship with the inherited friend classes. Set CAS of class ancestor classes, as defined in Transformation 12, is needed to derive set CIF.

The set CIF of classes with their inherited friend classes is defined as follows.

$$CIF = \{(ci, friend\_ci)\} = \{(x.ci, y.friend\_ci) \mid x \in CAS \wedge y \in CEF \wedge x.ancestor\_ci = y.ci\ \}$$

2.14. Transformation 14 - *FIF*

If a child class inherits from ancestor classes that have friend global functions defined then the child class has a partial friend relationship with the inherited friend global function. Set CAS of class ancestor classes, as defined in Transformation 12, is needed to derive set FIF.

The set FIF of classes with their inherited friend global functions is defined as follows.

$$FIF = \{(ci, friend\_fi)\} = \{(x.ci, y.friend\_fi) \mid x \in CAS \wedge y \in FF \wedge x.ancestor\_ci = y.ci\}$$

3. Appendix 3 – eMulePlus C++ Class Modularity to Lines of Code Correlation Data

The following table lists, for each class modularity measure taken from the eMulePlus software system, the correlation between the normalised measured value and a normalised count of the number of class lines of code. Missing values occur when the measured values for a measure are all the same and so cannot be normalised.

Table Appendix 3-1 eMulePlus software system class modularity measure to class lines of code correlation

			Zscore: Class Total Lines of Code
Spearman's rho	Zscore: CER1	Correlation Coefficient	.183(**)
		Sig. (2-tailed)	.004
		N	250
	Zscore: CER2	Correlation Coefficient	.187(**)
		Sig. (2-tailed)	.003
		N	250
	Zscore: CER3	Correlation Coefficient	.057
		Sig. (2-tailed)	.368
		N	250
	Zscore: CER4	Correlation Coefficient	.004
		Sig. (2-tailed)	.951
		N	250
	Zscore: CER5	Correlation Coefficient	.029
		Sig. (2-tailed)	.653
		N	250
	Zscore: CER6	Correlation Coefficient	.
		Sig. (2-tailed)	.
		N	0
	Zscore: CER7	Correlation Coefficient	.136(*)
		Sig. (2-tailed)	.032
		N	250
	Zscore: CUR1	Correlation Coefficient	.136(*)
		Sig. (2-tailed)	.032
		N	250
	Zscore: CUR2	Correlation Coefficient	.430(**)
		Sig. (2-tailed)	.000
		N	250
	Zscore: CDC1	Correlation Coefficient	.042
		Sig. (2-tailed)	.505
		N	250
	Zscore: CDC2	Correlation Coefficient	.172(**)
		Sig. (2-tailed)	.006
		N	250

	Zscore: CUER1	Correlation Coefficient	.042
		Sig. (2-tailed)	.505
		N	250
	Zscore: CUER2	Correlation Coefficient	.048
		Sig. (2-tailed)	.451
		N	250
	Zscore: CUER3	Correlation Coefficient	.119
		Sig. (2-tailed)	.061
		N	250
	Zscore: CNI1	Correlation Coefficient	.042
		Sig. (2-tailed)	.505
		N	250
	Zscore: CNI2	Correlation Coefficient	.167(**)
		Sig. (2-tailed)	.008
		N	250
	Zscore: CNI3	Correlation Coefficient	.066
		Sig. (2-tailed)	.298
		N	250
	Zscore: CSI1	Correlation Coefficient	.430(**)
		Sig. (2-tailed)	.000
		N	250
	Zscore: CSI2	Correlation Coefficient	.460(**)
		Sig. (2-tailed)	.000
		N	250
	Zscore: CEVR1	Correlation Coefficient	.110
		Sig. (2-tailed)	.082
		N	250
	Zscore: CEVR2	Correlation Coefficient	.145(*)
		Sig. (2-tailed)	.022
		N	250
	Zscore: CEFW4	Correlation Coefficient	.066
		Sig. (2-tailed)	.298
		N	250
	Zscore(ciei1)	Correlation Coefficient	.035
		Sig. (2-tailed)	.585
		N	250
	Zscore(ciei2)	Correlation Coefficient	-.042
		Sig. (2-tailed)	.513
		N	250
	Zscore(ciei3)	Correlation Coefficient	.
		Sig. (2-tailed)	.
		N	0
	Zscore(ciei4)	Correlation Coefficient	.
		Sig. (2-tailed)	.
		N	0
	Zscore(ciei5)	Correlation Coefficient	.
		Sig. (2-tailed)	.
		N	0

	Zscore(ciei6)	Correlation Coefficient	.
		Sig. (2-tailed)	.
		N	0
	Zscore(ciei7)	Correlation Coefficient	.185(**)
		Sig. (2-tailed)	.003
		N	250
	Zscore(ciei8)	Correlation Coefficient	.015
		Sig. (2-tailed)	.809
		N	250
	Zscore: COMPUTE cis12rat = cis1 / (cis1 + cis2) (COMPUTE)	Correlation Coefficient	.224(**)
		Sig. (2-tailed)	.000
		N	250
	Zscore: COMPUTE cis34rat = cis3 / (cis3 + cis4) (COMPUTE)	Correlation Coefficient	.044
		Sig. (2-tailed)	.486
		N	250
	Zscore(cis5)	Correlation Coefficient	.604(**)
		Sig. (2-tailed)	.000
		N	250
	Zscore(cis6)	Correlation Coefficient	.267(**)
		Sig. (2-tailed)	.000
		N	250
	Zscore(cis7)	Correlation Coefficient	.042
		Sig. (2-tailed)	.504
		N	250
	Zscore(cis8)	Correlation Coefficient	-.123
		Sig. (2-tailed)	.052
		N	250
	Zscore: CDE1	Correlation Coefficient	.254(**)
		Sig. (2-tailed)	.000
		N	250
	Zscore(cde2)	Correlation Coefficient	.042
		Sig. (2-tailed)	.504
		N	250
	Zscore(cde3)	Correlation Coefficient	-.123
		Sig. (2-tailed)	.052
		N	250
	Zscore(cde4)	Correlation Coefficient	.425(**)
		Sig. (2-tailed)	.000
		N	250
	Zscore(cde5)	Correlation Coefficient	.418(**)
		Sig. (2-tailed)	.000
		N	250
	Zscore: Class Total Lines of Code	Correlation Coefficient	1.000
		Sig. (2-tailed)	.
		N	250

** Correlation is significant at the 0.01 level (2-tailed).

* Correlation is significant at the 0.05 level (2-tailed).

4. Appendix 4 - eMulePlus Software System Content Validation

In step 2 of the content validation, importance and frequency ratings are assigned to each **char**acteristic to **me**asure **r**elationship (CHARMER) diagram software feature. The importance (I) and frequency (F) ratings used are as follows: 1- high, 2 - medium, 3 - low. In Figure 5-14, the importance and frequency ratings are shown in the natural language feature boxes beneath the feature identification number.

4.1. eMulePlus class modularity

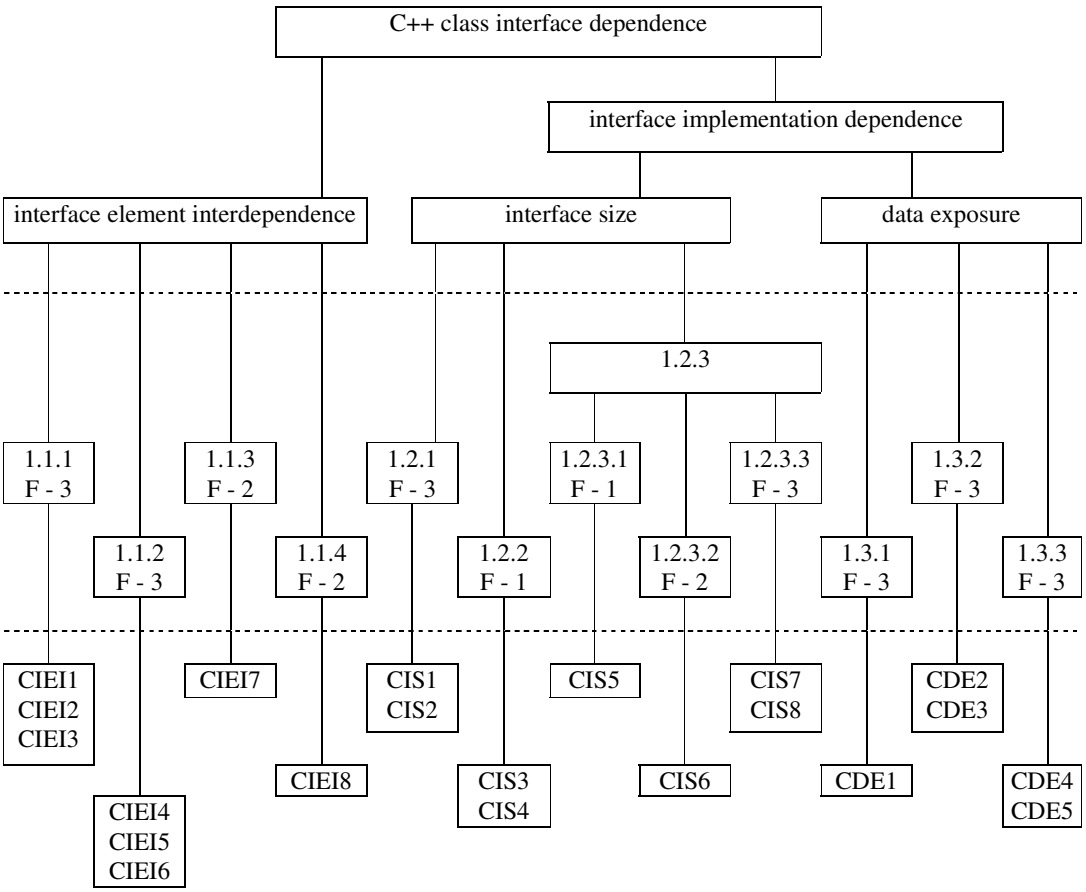


Figure Appendix 4-1 eMulePlus class interface dependence CHARMER diagram content validation

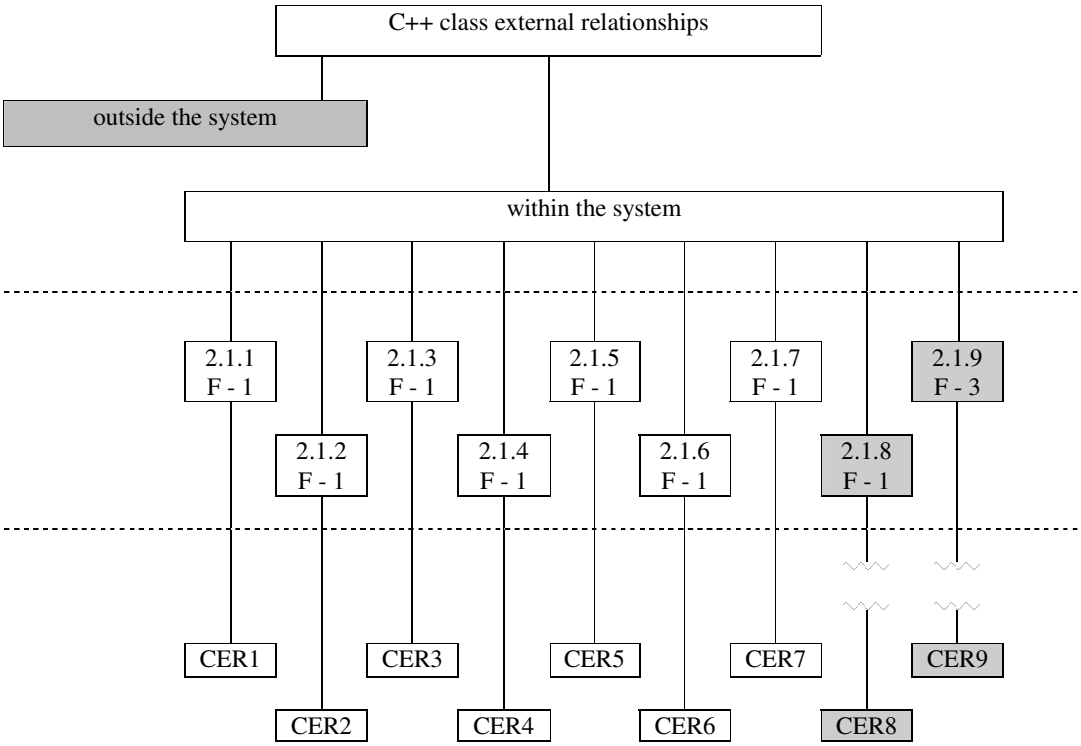


Figure Appendix 4-2 eMulePlus class external relationships CHARMER diagram content validation

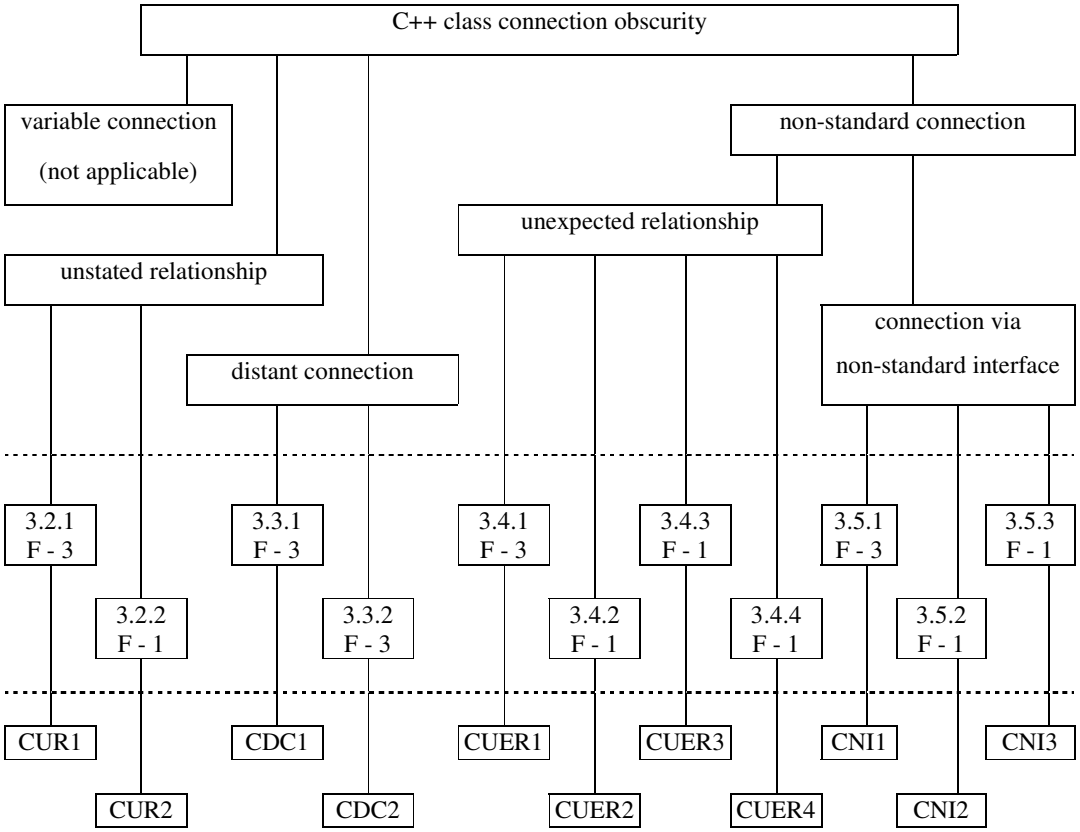


Figure Appendix 4-3 eMulePlus class connection obscurity CHARMER diagram content validation

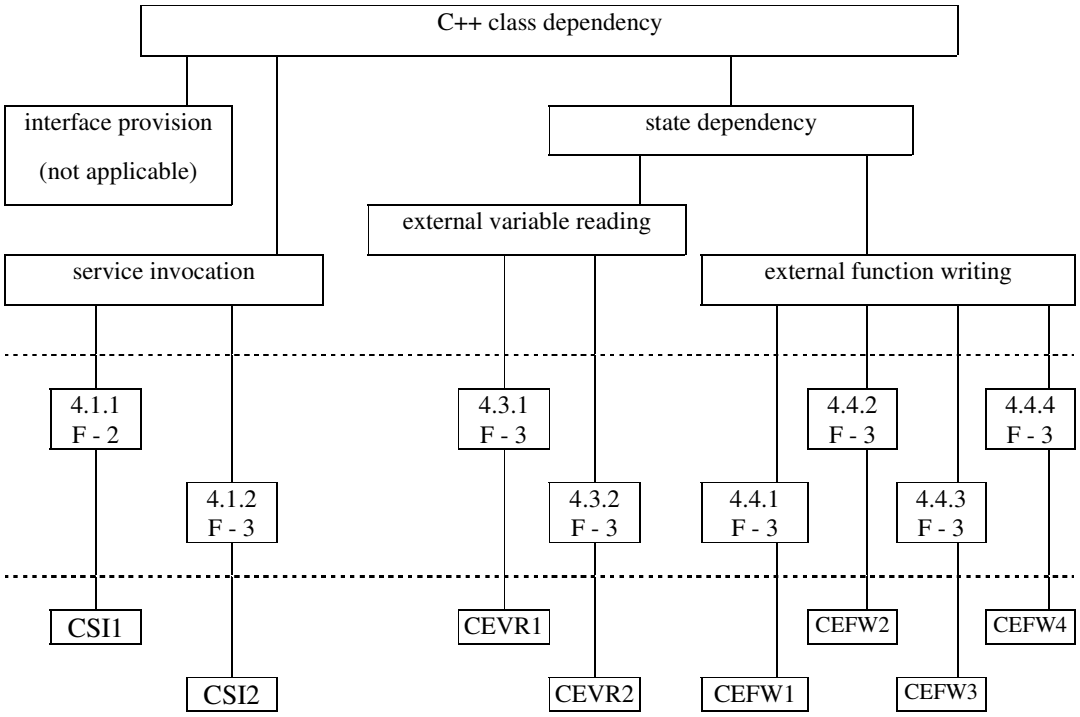


Figure Appendix 4-4 eMulePlus class dependency CHARMER diagram content validation

4.2. eMulePlus object modularity

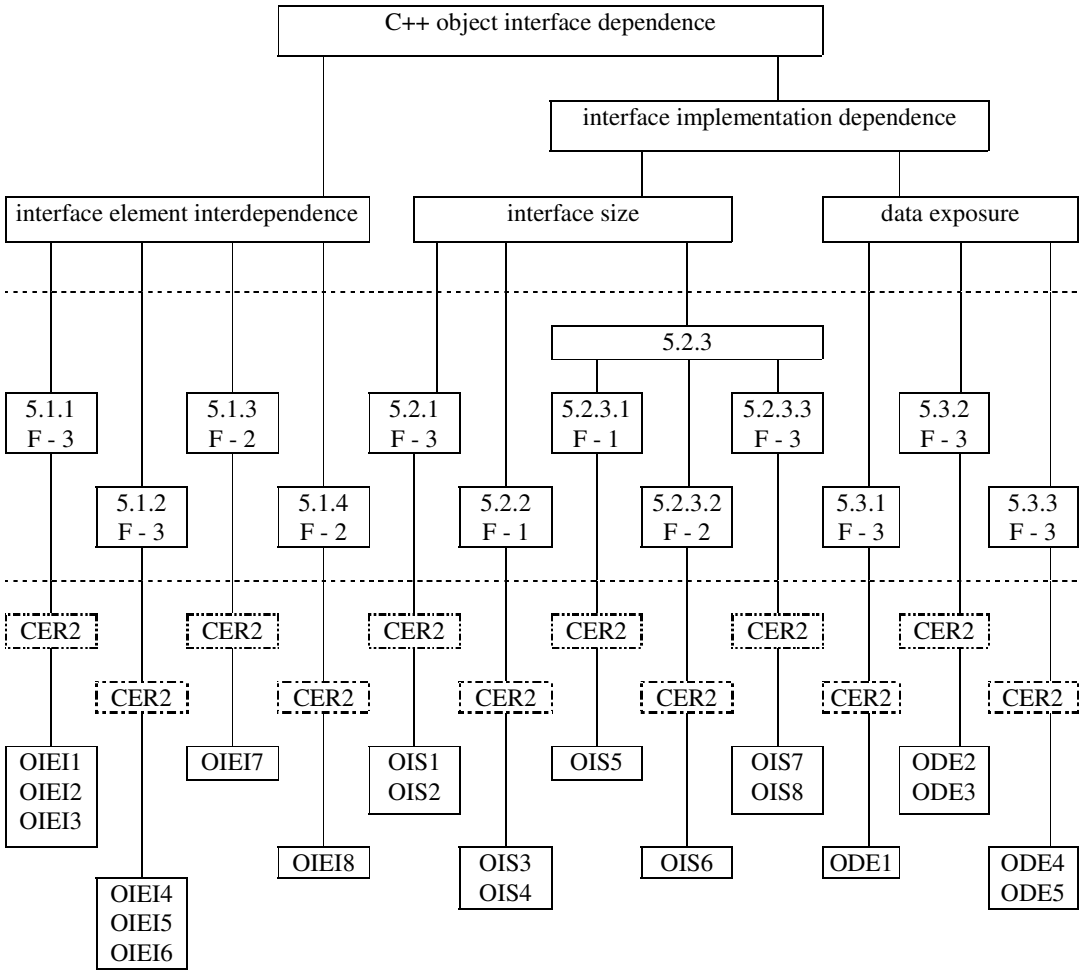


Figure Appendix 4-5 eMulePlus object interface dependence CHARMER diagram content validation

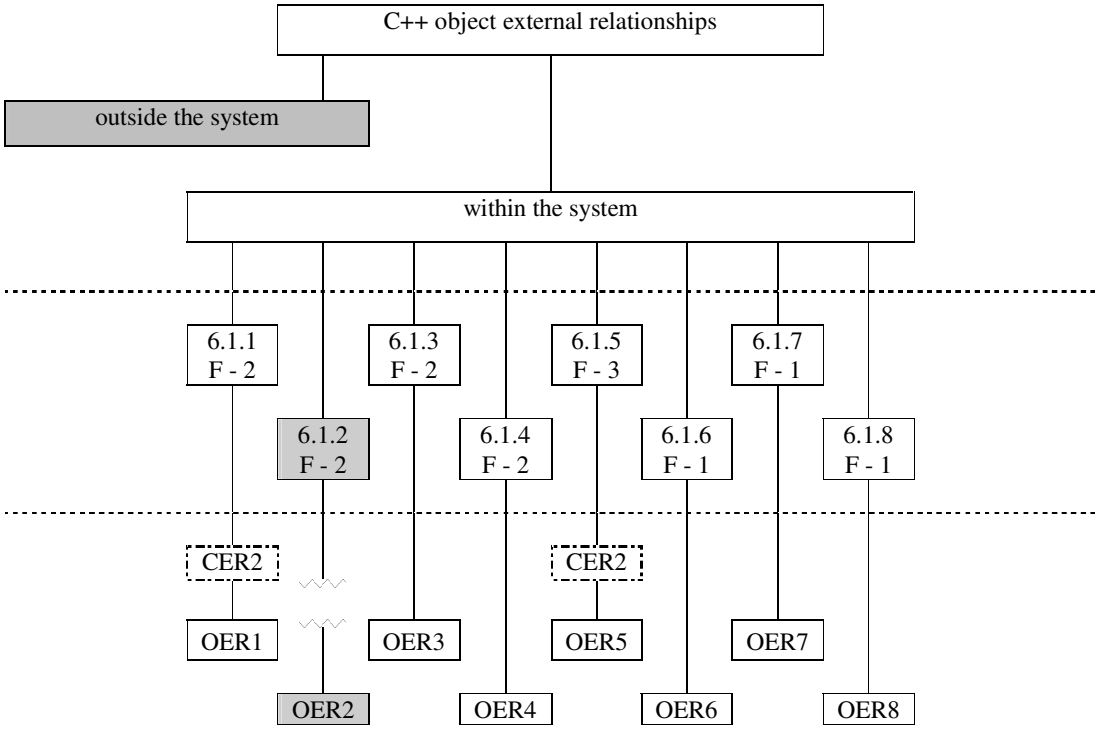


Figure Appendix 4-6 eMulePlus object external relationships CHARMER diagram content validation

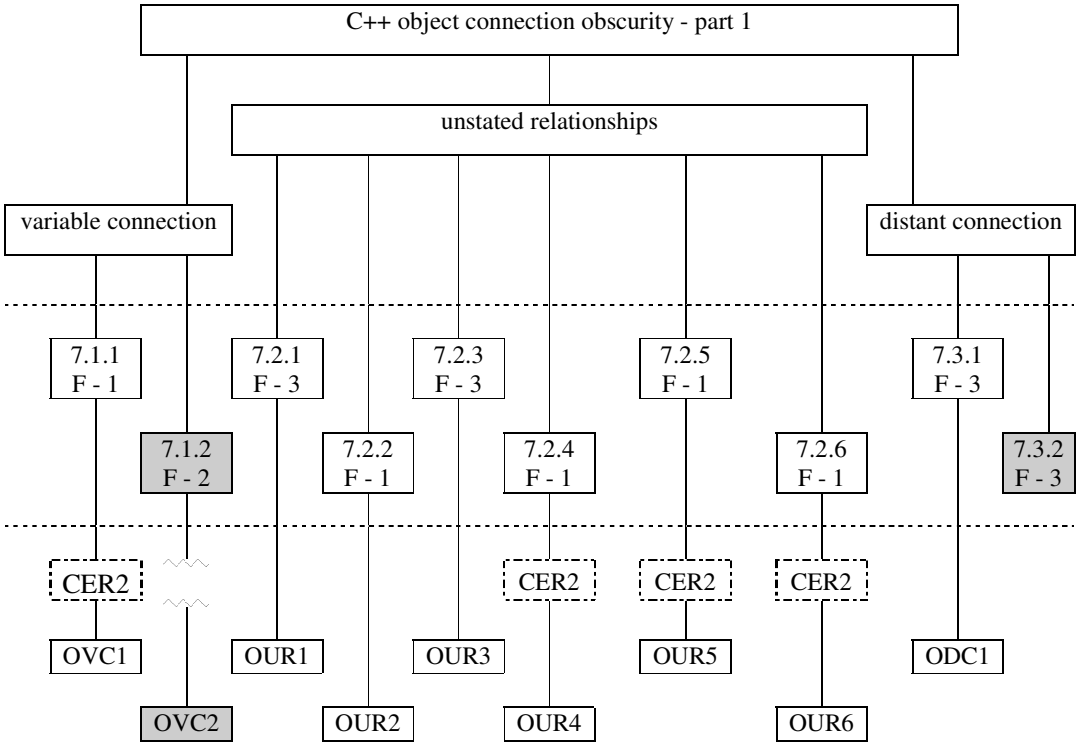


Figure Appendix 4-7 eMulePlus object connection obscurity CHARMER diagram content validation - Part 1

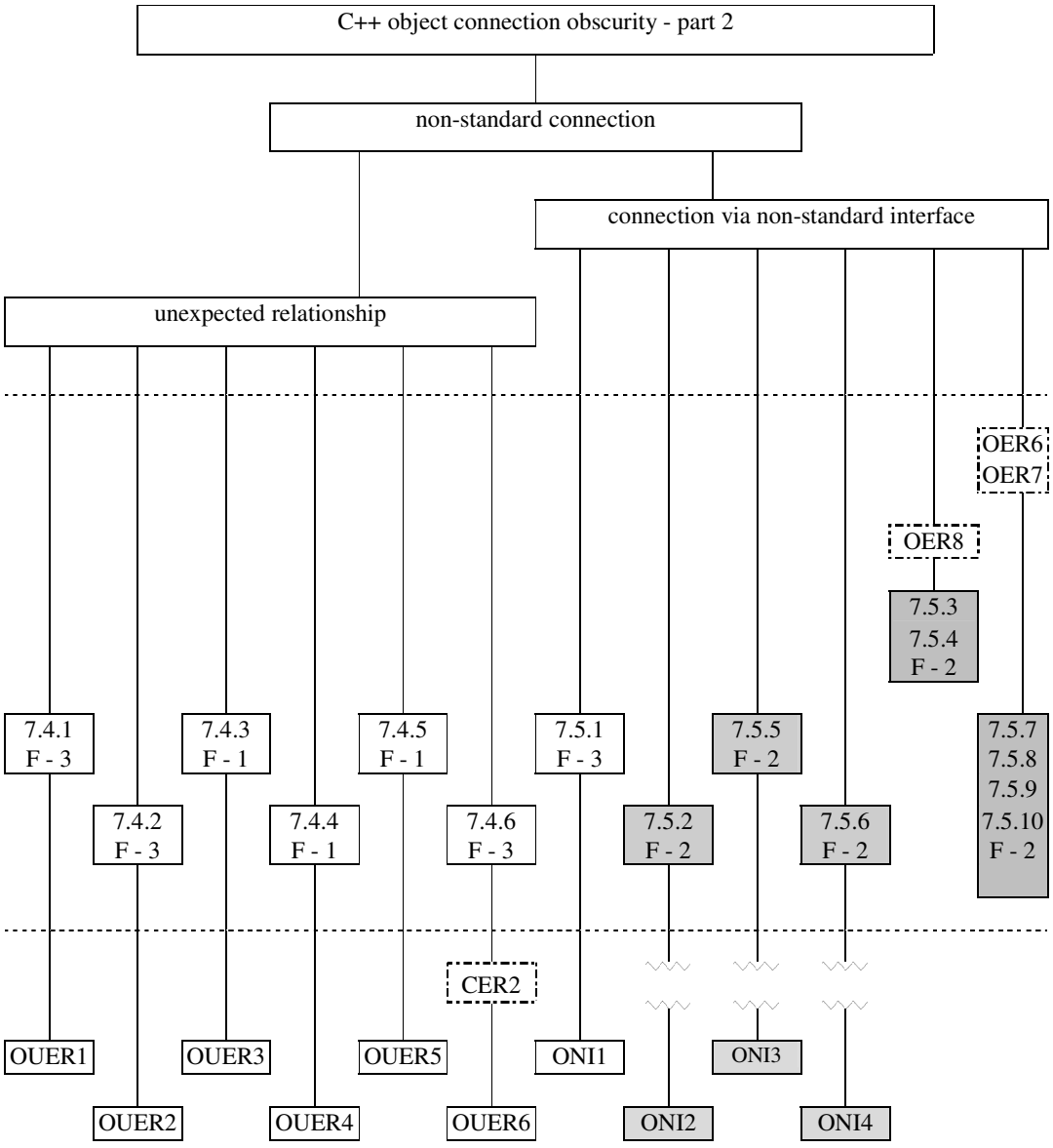


Figure Appendix 4-8 eMulePlus object connection obscurity CHARMER diagram content validation - Part 2

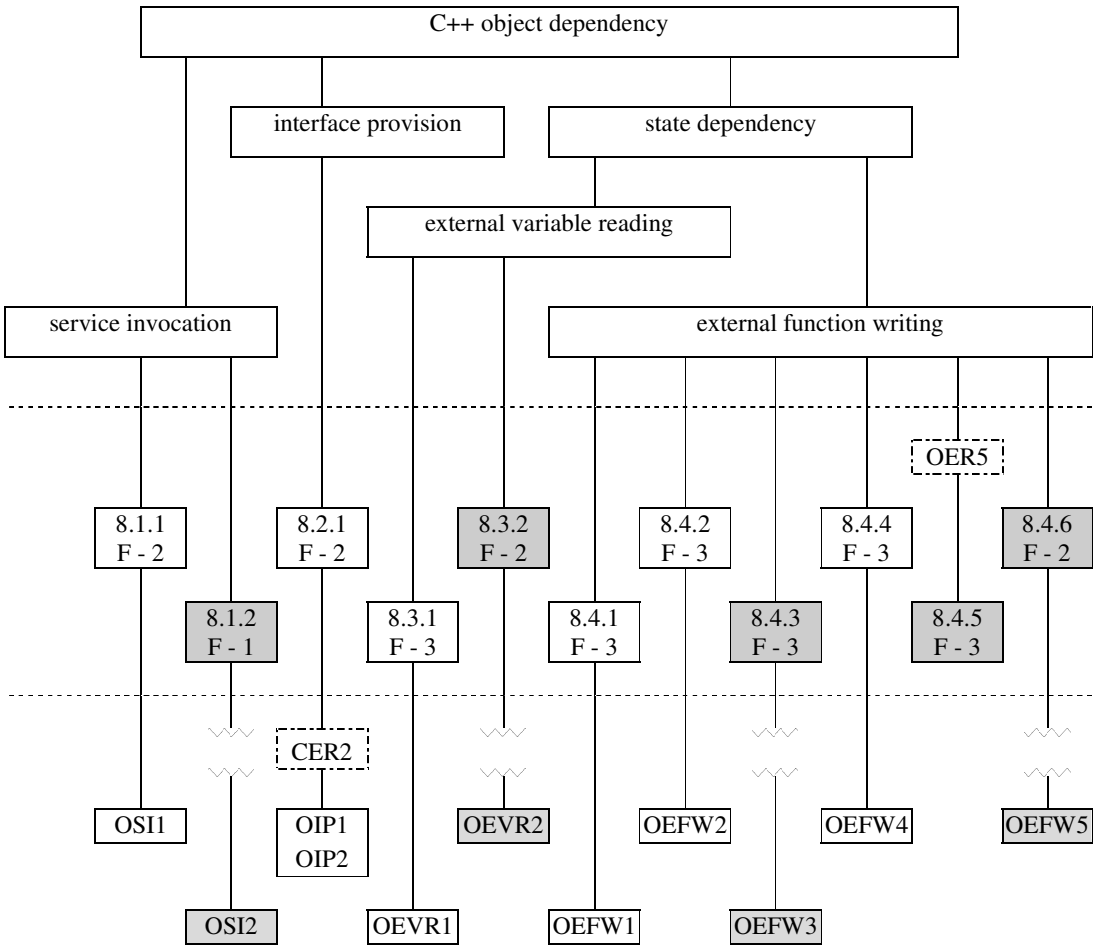


Figure Appendix 4-9 eMulePlus object dependency CHARMER diagram content validation

4.3. eMulePlus content validation measured values

The following table lists the eMulePlus content validation measured values for all the measured classes/object-classes in the eMulePlus system.

Table Appendix 4-1 eMulePlus software system content validation measured values

eMulePlus class/object-class	CER2	OER5	OER6	OER7	OER8
C3DPreviewControl	0	0	0	0	0
CAboutDlg	0	0	0	0	0
CAbstractFile	0	0	0	0	0
CAddFileThread	0	0	0	0	1
CAddFriend	0	0	0	0	0
CArchiveRecovery	0	0	0	0	0
CArrowCombo	0	0	0	0	0
CAsyncProxySocket	1	0	0	0	0
CAsyncProxySocketLayer	0	0	0	0	0
CAsyncSocketEx	0	0	2	0	1
CAsyncSocketExHelperWindow	0	0	0	0	4
CAsyncSocketExLayer	0	0	2	0	3
CBarShader	0	0	0	0	0
CButtonST	0	0	0	0	0
CCeXDib	0	0	0	0	0
CChatItem	0	0	0	0	0
CChatSelector	0	0	0	0	0
CChatWnd	4	0	0	0	0
CCKey	0	0	0	0	0
CClientCredits	0	0	0	0	0
CClientCreditsList	0	0	0	0	0
CClientDetailDialog	0	0	0	0	0
CClientList	0	0	0	0	0
CClientReqSocket	2	0	3	0	1
CClientSource	0	0	0	0	0
CClientUDPSocket	0	0	0	0	0
CClosableTabCtrl	0	0	0	0	0
CColorButton	0	0	0	0	0
CColorFrameCtrl	0	0	0	0	0
CColourPopup	0	0	0	0	0
CCommentDialog	0	0	0	0	0
CCommentDialogLst	4	0	0	0	0
CCreditsCtrl	0	0	0	0	0
CCriticalSection_INL	0	0	0	0	0
CDblScope	0	0	0	0	0
CDialogMinTrayBtn	0	0	0	0	0
CDirectoryTreeCtrl	0	0	0	0	0
CDownloadListCtrl	0	0	0	0	0
CDownloadQueue	0	0	2	0	0
CED2KFileLink	0	0	0	0	0
CED2KLink	0	0	0	0	0
CED2KServerLink	0	0	0	0	0
CED2KServerListLink	0	0	0	0	0
CEdit2	0	0	0	0	0

CEMSocket	1	0	0	0	0
CemuleApp	0	0	0	0	0
CemuleDlg	1	0	2	0	0
CEnBitmap	0	0	0	0	0
CFileDetailDialog	0	0	0	0	0
CFileHashControl	0	0	0	0	0
CFileInfoDialog	0	0	0	0	0
CFileStatistic	0	0	3	0	1
CFontPreviewCombo	0	0	0	0	0
CFriend	0	1	0	0	0
CFriendList	0	0	0	0	0
CFriendListCtrl	0	0	0	0	0
CGDIThread	0	0	0	0	0
CGradientStatic	0	0	0	0	0
CHostnameSourceWnd	0	0	0	0	0
CHttpDownloadDlg	0	0	0	0	0
CHyperLink	0	0	1	0	0
CHyperTextCtrl	0	0	0	0	0
CIconStatic	0	0	0	0	0
CInfoListCtrl	0	0	0	0	0
CIni	0	0	0	0	0
CInputBox	0	0	0	0	0
CIPFilter	0	0	0	0	0
CIrcMain	0	0	0	0	0
CIrcSocket	1	0	0	0	0
CIrcWnd	4	0	0	0	0
CKeyWord	0	0	1	0	0
CKnownFile	0	0	0	0	1
CKnownFileList	0	0	2	0	1
CLanCast	0	0	0	0	0
ClientsData	0	0	0	0	0
CLineInfo	0	0	0	0	0
CLinePartInfo	0	0	0	0	0
CListBoxST	0	0	0	0	0
CListCtrlSorter	0	0	0	0	1
CListCtrlSorterItem	0	0	1	0	0
CListenSocket	1	0	0	0	1
CLoggable	0	0	0	0	0
CMemDC	0	0	0	0	0
CMemDC2	0	0	0	0	0
CMeterIcon	0	0	0	0	0
CMuleCtrlItem	0	0	0	0	0
CMuleListCtrl	0	0	0	0	0
CMuleRollup	0	0	0	0	0
CMuleSystrayDlg	0	0	0	0	0
CMuleToolbarCtrl	0	0	0	0	1
CMyFont	0	0	0	0	0
CNewServerDlg	0	0	0	0	0
COptionTree	0	0	0	0	0
COptionTreeCheckButton	0	0	0	0	0
COptionTreeColorPopUp	0	0	0	0	0
COptionTreeFileDlg	0	0	0	0	0
COptionTreeFontSel	0	0	0	0	0
COptionTreeFontSelColorButton	0	0	0	0	0
COptionTreeFontSelFontCombo	0	0	0	0	0
COptionTreeFontSelSizeCombo	0	0	0	0	0

COptionTreeImagePopUp	0	0	0	0	0
COptionTreeInfo	0	0	0	0	0
COptionTreeIPAddressEdit	0	0	0	0	0
COptionTreeItem	0	0	0	0	0
COptionTreeItemCheckBox	0	0	0	0	0
COptionTreeItemColor	0	0	0	0	0
COptionTreeItemComboBox	0	0	0	0	0
COptionTreeItemDate	0	0	0	0	0
COptionTreeItemEdit	0	0	0	0	0
COptionTreeItemFile	0	0	0	0	0
COptionTreeItemFont	0	0	0	0	0
COptionTreeItemHyperLink	0	0	0	0	0
COptionTreeItemImage	0	0	0	0	0
COptionTreeItemIPAddress	0	0	0	0	0
COptionTreeItemRadio	0	0	0	0	0
COptionTreeItemSpinner	0	0	0	0	0
COptionTreeItemStatic	0	0	0	0	0
COptionTreeList	0	0	0	0	0
COptionTreeRadioButton	0	0	0	0	0
COptionTreeSpinnerButton	0	0	0	0	0
COptionTreeSpinnerEdit	0	0	0	0	0
COScopeCtrl	0	0	0	0	0
CPageSelectionBox	0	0	0	0	0
CPartFile	2	0	0	0	1
CPIDL	0	0	0	0	0
CPPgAdvanced	0	0	0	0	0
CPPgConnection	0	0	0	0	0
CPPgDirectories	0	0	0	0	0
CPPgFiles	0	0	0	0	0
CPPgGeneral	0	0	0	0	0
CPPgHTTPD	0	0	0	0	0
CPPgIRC	0	0	0	0	0
CPPgMessaging	0	0	0	0	0
CPPgModPT	0	0	0	0	0
CPPgNotify	0	0	0	0	0
CPPgProxy	0	0	0	0	0
CPPgScheduler	0	0	0	0	0
CPPgServer	0	0	0	0	0
CPPgSorting	0	0	0	0	0
CPPgStats	0	0	0	0	0
CPPgWindow	0	0	0	0	0
CPPToolTip	0	0	0	0	0
CPreferences	0	0	0	0	0
CPreferencesDlg	0	0	0	0	0
CPreparedHyperText	0	0	0	0	2
CPreviewThread	0	0	0	0	0
CProcessingCmdThread	0	0	0	0	0
CProgressCtrlX	0	0	0	0	0
CQArray	0	0	0	0	0
CQueueListCtrl	0	0	0	0	0
CResizableDialog	0	0	0	0	0
CResizableFormView	0	0	0	0	0
CResizableFrame	0	0	0	0	0
CResizableGrip	0	0	0	0	0
CResizableLayout	0	0	0	0	0
CResizableMDIChild	0	0	0	0	0

CResizableMDIFrame	0	0	0	0	0
CResizableMinMax	0	0	0	0	0
CResizablePage	0	0	0	0	0
CResizablePageEx	0	0	0	0	0
CResizableSheet	0	0	0	0	0
CResizableSheetEx	0	0	0	0	0
CResizableState	0	0	0	0	0
CRollupCtrl	0	0	0	0	0
CRollupGripper	0	0	0	0	0
CRollupHeader	0	0	0	0	0
CRWLockLite	0	0	0	0	0
CSafeArray	0	0	0	0	0
CSafeArraySorted	0	0	0	0	0
CSafeFile	0	0	0	0	0
CSafeMemFile	0	0	0	0	0
CSaveDC	0	0	0	0	0
CSearchDlg	4	0	0	0	0
CSearchFile	0	0	1	0	0
CSearchList	0	0	1	0	0
CSearchListCtrl	0	0	0	0	1
CSecuredVar	0	0	0	0	0
CSelBitmap	0	0	0	0	0
CSelBkColor	0	0	0	0	0
CSelBkMode	0	0	0	0	0
CSelBrush	0	0	0	0	0
CSelect	0	0	0	0	0
CSelFont	0	0	0	0	0
CSelMapMode	0	0	0	0	0
CSelPalette	0	0	0	0	0
CSelPen	0	0	0	0	0
CSelROP2	0	0	0	0	0
CSelStock	0	0	0	0	0
CSelTextAlign	0	0	0	0	0
CSelTextColor	0	0	0	0	0
CServer	0	0	0	0	0
CServerConnect	0	0	0	0	2
CServerEntry	0	0	0	0	0
CServerList	0	0	2	0	0
CServerListCtrl	0	0	0	0	1
CServerSocket	2	0	3	0	0
CServerWnd	4	0	1	0	0
CSharedFileList	0	0	3	0	0
CSharedFilesCtrl	0	0	0	0	2
CSharedFilesWnd	4	0	0	0	1
CShellContextMenu	0	0	0	0	0
CSourceEntry	0	0	0	0	0
CSplashScreen	0	0	0	0	0
CSplitterControl	0	0	0	0	0
CStatisticsData	0	0	0	0	0
CStatisticsDlg	4	0	0	0	0
CStoredSources	0	0	0	0	0
CStoredSourcesContainer	0	0	0	0	0
CTag	0	0	0	0	0
CTaskbarNotifier	0	0	0	0	0
CThemeHelperST	0	0	0	0	0
CTitleMenu	0	0	0	0	0

CTransferWnd	4	0	0	0	0
CTrayDialog	0	0	0	0	0
CTrayMenuBtn	0	0	0	0	0
CUDPSocket	0	0	1	0	0
CUDPSocketWnd	0	0	0	0	0
CUpdateServerMetDlg	0	0	0	0	1
CUpDownClient	0	0	1	0	0
CUploadListCtrl	0	0	0	0	0
CUploadQueue	0	0	1	0	1
CVisLine	0	0	0	0	0
CVisPart	0	0	0	0	0
CVisualStylesXP	0	0	0	0	0
CWebServer	0	0	1	0	5
CWebSocket	0	0	0	0	1
CXPStyleButtonST	0	0	0	0	0
Db	0	0	1	0	3
Dbc	0	0	1	0	1
DbDeadlockException	0	0	0	0	0
DbEnv	0	0	3	0	7
DbException	0	0	0	0	0
DbLock	0	0	1	0	1
DbLockNotGrantedException	0	0	0	0	0
DbLogc	0	0	1	0	2
DbLsn	0	0	2	0	0
DbMemoryException	0	0	0	0	0
DbMpoolFile	0	0	1	0	1
DbPreplist	0	0	0	0	0
DbRunRecoveryException	0	0	0	0	0
Dbt	0	0	4	0	0
DbTxn	0	0	1	0	0
InputBox	0	0	0	0	0
MD5Sum	0	0	0	0	0
MiniDumper	0	0	0	0	0
Packet	0	0	0	0	0
sfl_itemdata	0	0	0	0	0
StatusBarCtrl	0	0	0	0	0
XBMDraw	0	0	0	0	0

5. Appendix 5 – eMulePlus Modularity Data Values

The following table lists, for each measured eMulePlus software system class, the total lines of code, unweighted and weighted modularity aggregate and Euclidean distance values calculated according to the methods described in section 7.1.

Table Appendix 5-1 eMulePlus software system class lines of code, unweighted and weighted modularity aggregate and modularity Euclidean distance values

Case Summaries

class	eMulePlus Class Lines of Code	eMulePlus Class Modularity Aggregate	eMulePlus Weighted Class Modularity Aggregate	eMulePlus Class Modularity Euclidean Distance
C3DPreviewControl	16.00	-.87585	-.91668	2.07
CAboutDlg	29.00	-.60070	-.36159	3.60
CAbstractFile	37.00	2.58045	1.70363	18.94
CAddFileThread	27.00	-.28231	-.35316	4.25
CAddFriend	56.00	-.50632	-.41479	3.39
CArchiveRecovery	721.00	-.29650	-.36957	4.21
CArrowCombo	6.00	-.73739	-.63735	3.94
CAsyncProxySocket	780.00	.11448	.55587	5.88
CAsyncProxySocketLayer	1026.00	-.24634	-.02477	5.11
CAsyncSocketEx	649.00	1.12790	.74668	7.48
CAsyncSocketExHelperWindow	235.00	-.13913	-.21102	6.17
CAsyncSocketExLayer	411.00	.66231	.78323	7.28
CBarShader	212.00	-.53461	-.22825	5.17
CButtonST	1025.00	-.58651	-.41554	4.87
CCeXDib	234.00	-.75752	-.67797	3.61
CChatItem	5.00	-.82489	-1.05747	.96
CChatSelector	339.00	-.17973	-.47212	3.31
CChatWnd	42.00	.43906	.07436	5.93
CCKey	3.00	-.55626	-.51555	4.05
CClientCredits	68.00	-.67394	-.75294	3.23
CClientCreditsList	111.00	-.04057	-.07867	4.92
CClientDetailDialog	177.00	.41749	.23093	6.38
CClientList	238.00	-.62400	-.65221	3.53
CClientReqSocket	585.00	3.28632	2.53095	12.84
CClientSource	6.00	-.75634	-.67558	4.60
CClientUDPSocket	172.00	.11750	.35294	5.20
CClosableTabCtrl	58.00	-.59802	-.35619	3.53
CColorButton	196.00	3.77392	2.25091	23.08
CColorFrameCtrl	37.00	-.83842	-.84116	3.15
CColourPopup	578.00	.35790	1.57230	9.06
CCommentDialog	48.00	-.61272	-.62944	2.07

CCommentDialogLst	68.00	1.10557	.59235	7.27
CCreditsCtrl	969.00	-.06287	.72344	6.76
CCriticalSection_INL	6.00	-.86457	-.89393	3.07
CDblScope	148.00	-.54958	-.25847	4.14
CDialogMinTrayBtn	20.00	-.49566	-.40728	4.20
CDirectoryTreeCtrl	392.00	-.50785	-.66147	2.97
CDownloadListCtrl	1931.00	2.06766	1.49134	11.84
CDownloadQueue	883.00	1.04631	.56561	7.45
CED2KFileLink	153.00	-.25008	-.40680	4.03
CED2KLink	47.00	-.04337	-.06667	6.07
CED2KServerLink	49.00	-.43120	-.52859	3.67
CED2KServerListLink	35.00	-.64825	-.72288	3.41
CEdit2	15.00	-.88060	-.92627	2.00
CEMSocket	415.00	2.41215	1.61155	11.13
CemuleApp	383.00	1.69193	1.22043	10.87
CemuleDlg	1149.00	1.68495	1.23411	8.51
CEnBitmap	140.00	.56702	.60114	7.83
CFileDetailDialog	211.00	.02838	-.31046	5.13
CFileHashControl	136.00	-.80750	-.77880	3.37
CFileInfoDialog	159.00	-.30829	-.25887	4.27
CFileStatistic	314.00	1.23502	1.29067	11.05
CFontPreviewCombo	200.00	.51583	1.89091	10.35
CFriend	79.00	3.15043	6.60609	23.18
CFriendList	171.00	-.00270	-.13315	4.53
CFriendListCtrl	137.00	-.53307	-.58146	3.64
CGDIThread	45.00	-.60603	-.37234	4.45
CGradientStatic	140.00	-.64065	-.68579	3.39
CHostnameSourceWnd	3.00	-.99958	-1.16630	.00
CHttpDownloadDlg	537.00	-.64065	-.68579	3.39
CHyperLink	67.00	-.37111	-.36358	4.75
CHyperTextCtrl	755.00	-.50675	-.51020	2.93
CIconStatic	25.00	-.84555	-.85555	3.12
CInfoListCtrl	503.00	.19333	-.11217	5.28
CIni	661.00	-.27211	.30130	6.87
CInputBox	.00	-1.00601	-1.17927	.00
CIPFilter	122.00	-.83604	-.83637	3.16
CIrcMain	443.00	.09227	-.16698	4.62
CIrcSocket	132.00	.33393	-.05347	5.25
CIrcWnd	1159.00	1.59339	1.35106	8.59
CKeyWord	12.00	-.37111	-.36358	4.75
CKnownFile	788.00	4.40733	3.03664	15.83
CKnownFileList	337.00	.84524	.25687	6.42
CLanCast	188.00	-.02054	.09264	8.47
ClientsData	15.00	-1.00601	-1.17927	.00
CLineInfo	12.00	-.28369	.27795	5.56
CLinePartInfo	15.00	-.22230	.31923	5.58
CListBoxST	379.00	-.73177	-.62601	3.80

CListCtrlSorter	29.00	-.44699	-.55450	3.85
CListCtrlSorterItem	48.00	1.47849	3.36783	14.62
CListenSocket	142.00	.18672	.08588	5.57
CLoggable	24.00	1.81771	.98019	13.92
CMemDC	55.00	-.85030	-.86515	3.10
CMemDC2	59.00	-.85030	-.86515	3.10
CMeterIcon	292.00	-.14268	.56243	6.33
CMuleCtrlItem	12.00	-.86457	-.89393	3.07
CMuleListCtrl	630.00	-.09998	.15315	4.95
CMuleRollup	6.00	-.80871	-1.04659	1.37
CMuleSystrayDlg	344.00	-.11418	.37632	5.76
CMuleToolbarCtrl	453.00	-.00436	.20759	5.85
CMyFont	24.00	-.78848	-.74043	3.57
CNewServerDlg	43.00	-.63167	-.66768	3.37
COptionTree	828.00	-.61153	-.46601	4.15
COptionTreeCheckButton	144.00	-.78891	-.82386	3.13
COptionTreeColorPopUp	798.00	.56755	1.75166	9.23
COptionTreeFileDlg	257.00	-.82177	-.80758	3.25
COptionTreeFontSel	749.00	.58871	.64490	7.78
COptionTreeFontSelColorButton	208.00	3.78819	2.27969	23.09
COptionTreeFontSelFontCombo	103.00	-.45953	-.56399	3.77
COptionTreeFontSelSizeCombo	44.00	-.65016	-.70497	3.33
COptionTreeImagePopUp	340.00	.33196	1.27637	6.93
COptionTreeInfo	15.00	-.85030	-.86515	3.10
COptionTreeIPAddressEdit	32.00	-.83604	-.83637	3.16
COptionTreeItem	515.00	.83016	.60812	7.77
COptionTreeItemCheckBox	100.00	-.27383	-.45830	4.33
COptionTreeItemColor	278.00	-.46638	-.58138	3.64
COptionTreeItemComboBox	111.00	-.30465	-.14241	4.91
COptionTreeItemDate	118.00	-.55494	-.64733	3.49
COptionTreeItemEdit	426.00	-.40933	-.35359	3.77
COptionTreeItemFile	359.00	-.35485	-.24368	4.56
COptionTreeItemFont	314.00	-.11733	.12278	7.18
COptionTreeItemHyperLink	324.00	-.56920	-.67612	3.44
COptionTreeItemImage	423.00	-.20519	-.05447	5.08
COptionTreeItemIPAddress	287.00	-.50262	-.54180	3.91
COptionTreeItemRadio	136.00	-.27383	-.45830	4.33
COptionTreeItemSpinner	138.00	-.27383	-.45830	4.33
COptionTreeItemStatic	91.00	-.57396	-.68571	3.42
COptionTreeList	624.00	.06275	.48967	5.68
COptionTreeRadioButton	204.00	-.74611	-.73752	3.40
COptionTreeSpinnerButton	560.00	-.46912	-.17872	5.21
COptionTreeSpinnerEdit	162.00	-.54483	-.24888	4.17
COScopeCtrl	518.00	.00997	.87039	8.51
CPageSelectionBox	7.00	-.64646	-.45391	3.02
CPartFile	3543.00	5.23590	3.51692	22.61

CPIDL	171.00	-.64392	-.44879	3.65
CPPgAdvanced	177.00	-.12446	.11199	4.76
CPPgConnection	480.00	.21040	.67482	6.20
CPPgDirectories	171.00	-.32535	-.04970	4.27
CPPgFiles	217.00	-.01864	.56906	6.08
CPPgGeneral	204.00	-.02350	.18120	4.75
CPPgHTTPD	88.00	-.43189	-.26464	3.57
CPPgIRC	169.00	-.31604	-.03091	4.59
CPPgMessaging	82.00	-.52704	-.45659	2.77
CPPgModPT	132.00	-.30965	-.01801	4.68
CPPgNotify	133.00	-.30653	-.01173	4.60
CPPgProxy	108.00	-.71124	-.82819	2.37
CPPgScheduler	208.00	-.31034	-.01941	4.44
CPPgServer	150.00	-.31398	-.02677	4.62
CPPgSorting	321.00	-.23154	.13956	4.28
CPPgStats	120.00	-.04942	.50697	5.65
CPPgWindow	119.00	-.12942	.10198	4.95
CPPToolTip	2134.00	.05035	.95185	8.56
CPreferences	1594.00	-.03118	-.16887	4.97
CPreferencesDlg	161.00	-.48596	-.37372	3.09
CPreparedHyperText	674.00	-.04286	-.24221	5.57
CPreviewThread	90.00	-1.00601	-1.17927	.00
CProcessingCmdThread	182.00	-.55370	-.62308	3.18
CProgressCtrlX	369.00	-.19037	-.09734	5.35
CQArray	6.00	-.65967	-.72416	3.28
CQueueListCtrl	569.00	.21813	-.19659	4.53
CResizableDialog	61.00	.75523	.20106	7.34
CResizableFormView	71.00	.09895	-.08430	5.16
CResizableFrame	20.00	-.36240	-.52425	4.20
CResizableGrip	80.00	.72166	.57563	9.60
CResizableLayout	336.00	.82954	.62813	8.56
CResizableMDIChild	31.00	-.22392	-.24490	4.40
CResizableMDIFrame	20.00	-.36240	-.52425	4.20
CResizableMinMax	76.00	.63007	.14315	8.43
CResizablePage	20.00	-.63517	-.80919	2.19
CResizablePageEx	27.00	-.61912	-.77682	2.42
CResizableSheet	176.00	1.36428	.58459	8.90
CResizableSheetEx	197.00	.24033	-.17715	6.25
CResizableState	57.00	.66141	.28895	8.70
CRollupCtrl	315.00	-.55068	-.34326	4.51
CRollupGripper	105.00	1.37146	3.61707	14.89
CRollupHeader	155.00	-.69319	-.54819	4.11
CRWLockLite	5.00	-.86457	-.89393	3.07
CSafeArray	349.00	-.78378	-.81352	2.82
CSafeArraySorted	54.00	-.56445	-.66652	3.45
CSafeFile	9.00	-.57336	-.30644	4.04
CSafeMemFile	9.00	-.57336	-.30644	4.04

CSaveDC	10.00	-.13655	.30944	6.58
CSearchDlg	956.00	2.26987	1.49423	9.68
CSearchFile	82.00	-.14029	-.40687	4.29
CSearchList	206.00	.84864	.52287	7.72
CSearchListCtrl	283.00	.51355	.14355	6.11
CSecuredVar	54.00	-.81464	-.79319	3.31
CSelBitmap	18.00	-.64350	-.71328	3.42
CSelBkColor	17.00	-.64350	-.71328	3.42
CSelBkMode	15.00	.21357	1.01578	10.64
CSelBrush	17.00	-.64350	-.71328	3.42
CSelect	2.00	-.20790	-.64256	5.54
CSelFont	17.00	-.64350	-.71328	3.42
CSelMapMode	15.00	.21357	1.01578	10.64
CSelPalette	32.00	-.63874	-.70369	3.45
CSelPen	17.00	-.64350	-.71328	3.42
CSelROP2	15.00	.21357	1.01578	10.64
CSelStock	15.00	-.64825	-.72288	3.41
CSelTextAlign	17.00	-.64350	-.71328	3.42
CSelTextColor	17.00	-.64350	-.71328	3.42
CServer	255.00	-.34932	-.45436	3.95
CServerConnect	458.00	.72201	.45376	6.57
CServerEntry	66.00	-.26328	.31913	5.70
CServerList	714.00	.69978	.37187	6.88
CServerListCtrl	581.00	.45631	.13720	6.20
CServerSocket	320.00	1.73115	1.00624	9.72
CServerWnd	260.00	1.80946	.94373	8.55
CSharedFileList	230.00	1.00181	.38515	8.41
CSharedFilesCtrl	1505.00	2.23561	2.24940	13.67
CSharedFilesWnd	133.00	1.71399	1.31678	8.38
CShellContextMenu	182.00	-.63114	-.66660	3.46
CSourceEntry	405.00	-.31457	.10294	6.75
CSplashScreen	65.00	-.56761	-.29483	3.81
CSplitterControl	230.00	-.01501	.81998	7.33
CStatisticsData	208.00	-.81251	-.78890	3.22
CStatisticsDlg	688.00	2.09415	1.28955	9.80
CStoredSources	455.00	-.32585	.08018	6.59
CStoredSourcesContainer	78.00	-.57723	-.42696	4.22
CTag	145.00	-.58834	-.58025	3.92
CTaskbarNotifier	529.00	-.49288	-.38768	3.59
CThemeHelperST	59.00	-.79253	-.74859	3.14
CTitleMenu	109.00	-.57346	-.30664	4.89
CTransferWnd	475.00	.78168	.63469	7.49
CTrayDialog	237.00	-.32944	-.16228	4.89
CTrayMenuBtn	79.00	-.58819	-.33635	4.66
CUDPSocket	251.00	1.00840	1.57237	10.72
CUDPSocketWnd	2.00	-1.00601	-1.17927	.00
CUpdateServerMetDlg	43.00	-.34386	-.47733	3.95

CUpDownClient	2620.00	1.74706	1.15012	9.53
CUploadListCtrl	506.00	.28606	-.15051	4.78
CUploadQueue	644.00	1.56944	.81678	9.65
CVisLine	.00	-1.00601	-1.17927	.00
CVisPart	15.00	-.66111	-.74882	2.98
CVisualStylesXP	420.00	-.79305	-.74964	3.10
CWebServer	2165.00	2.52829	1.33826	12.85
CWebSocket	166.00	1.70375	3.89715	16.80
CXPStyleButtonST	56.00	-.56920	-.67612	3.44
Db	18.00	.53495	.29343	8.05
Dbc	.00	-.36995	-.75154	3.23
DbDeadlockException	.00	-.80871	-1.04659	1.37
DbEnv	18.00	2.21865	1.19864	16.63
DbException	.00	-.76044	-1.01413	1.71
DbLock	.00	-.36995	-.75154	3.23
DbLockNotGrantedException	.00	-.80871	-1.04659	1.37
DbLogc	.00	-.07976	-.55639	4.84
DbLsn	.00	-.31429	-.71411	4.95
DbMemoryException	.00	-.80871	-1.04659	1.37
DbMpoolFile	10.00	-.02688	-.05943	4.83
DbPreplist	.00	-.87882	-.92269	2.46
DbRunRecoveryException	.00	-.80871	-1.04659	1.37
Dbt	17.00	.51888	.03640	10.37
DbTxn	18.00	-.50444	-.63257	3.97
InputBox	22.00	-.71712	-.84006	2.29
MD5Sum	28.00	-.27841	-.44219	4.33
MiniDumper	128.00	-.53262	-.71145	2.63
Packet	158.00	.33794	1.53203	10.11
sfl_itemdata	.00	-.85841	-.88150	2.65
StatusBarCtrl	70.00	-.52176	-.44594	4.93
XBMDraw	168.00	-.26137	.32297	7.77

The following tables list the eMulePlus object-class modularity aggregates derived in the manner described in section 7.1 of Chapter 7. As shown in Table 7-1 of Chapter 7, due to limitations of the measurement instrument, the full object modularity aggregation cannot be calculated. This is due to the inability of the measurement instrument to measure with sufficient validity, the connection via non-standard interface sub-characteristic of object connection obscurity and the entire dependency sub-characteristic. In place of the full object modularity aggregate, aggregates are calculated for the major measured object modularity sub-characteristics and from these, a partial object modularity aggregate is calculated.

Table Appendix 5-2 eMulePlus software system class and object modularity aggregate values

Class	eMulePlus Object Interface Dependence Aggregate	eMulePlus Object External Relationships Aggregate	eMulePlus Object Connection Obscurity Aggregate	eMulePlus Partial Object Modularity Aggregate
C3DPreviewControl	-.35368	-.33725	-.53396	-.59684
CAboutDlg	.91996	-.46311	-.53396	-.03757
CAbstractFile	.23408	-.49163	-.53396	-.38567
CAddFileThread	-.22217	.02011	.14459	-.02800
CAddFriend	-.65407	-.49163	.22639	-.44794
CArchiveRecovery	-.45693	-.49163	.22639	-.35188
CArrowCombo	.24574	-.36577	-.53396	-.31866
CAsyncProxySocket	-.15834	-.40606	1.66563	.53658
CAsyncProxySocketLayer	.03224	.09739	1.66563	.87475
CAsyncSocketEx	-.02863	3.22276	1.35366	2.21595
CAsyncSocketExHelperWindow	.01374	1.32715	.88996	1.08700
CAsyncSocketExLayer	-.65407	3.16621	1.07258	1.74669
CBarShader	-.28160	-.36577	.12806	-.25304
CButtonST	-.17025	-.49163	-.53396	-.58268
CCeXDib	-.17500	-.36577	-.53396	-.52367
CChatItem	-.65407	-.43459	-.04589	-.55282
CChatSelector	-.43376	-.33725	-.43563	-.58795
CChatWnd	-.65407	-.30872	-.43563	-.68139
CCKey	.24574	-.49163	-.43563	-.33208
CClientCredits	-.33358	-.36577	-.43563	-.55303
CClientCreditsList	-.28168	-.33725	-.24076	-.41889
CClientDetailDialog	-.65407	-.46311	.25088	-.42211
CClientList	-.20562	-.36577	-.43563	-.49068
CClientReqSocket	-.35288	1.90174	3.80299	2.60773
CClientSource	.61765	-.49163	-.53396	-.19878
CClientUDPSocket	-.36228	-.36577	-.43563	-.56701
CClosableTabCtrl	.90174	-.36577	-.53396	.00098
CColorButton	-.43496	.60017	7.51239	3.74097
CColorFrameCtrl	-.42680	-.36577	-.53396	-.64636
CColourPopup	3.04718	-.49163	-.53396	.98504
CCommentDialog	-.65407	-.46311	-.24076	-.66167

CCommentDialogLst	-.65407	-.46311	-.14243	-.61376
CCreditsCtrl	1.02286	-.36577	-.53396	.05999
CCriticalSection_INL	-.35795	-.36577	-.53396	-.61282
CDblScope	-.29734	-.36577	-.53396	-.58328
CDialogMinTrayBtn	-.49175	-.49163	-.53396	-.73934
CDirectoryTreeCtrl	-.40900	-.33725	-.33730	-.52797
CDownloadListCtrl	-.14108	-.36577	.44932	-.02803
CDownloadQueue	-.20955	.97814	1.28421	1.00025
CED2KFileLink	-.27234	-.49163	-.33730	-.53661
CED2KLink	1.04796	-.49163	-.43563	.05881
CED2KServerLink	-.27234	-.49163	-.43563	-.58452
CED2KServerListLink	-.30920	-.49163	-.43563	-.60248
CEdit2	-.36586	-.36577	-.53396	-.61667
CEMSocket	-.32872	-.34901	2.25203	.76709
CemuleApp	.76342	.02180	3.83419	2.25085
CemuleDlg	-.10892	1.10900	.40640	.68532
CEnBitmap	.38386	-.23991	.65232	.38799
CFileDetailDialog	-.46088	-.43459	.34742	-.26704
CFileHashControl	-.25836	-.33725	-.33909	-.45544
CFileInfoDialog	-.65407	-.46311	-.14243	-.61376
CFileStatistic	.76011	1.85646	.52714	1.53180
CFontPreviewCombo	1.93565	.60017	.12806	1.29800
CFriend	8.67226	4.92418	2.06161	7.62951
CFriendList	-.16124	-.33725	-.14243	-.31229
CFriendListCtrl	-.38456	-.21139	-.43563	-.50264
CGDIThread	.44554	-.49163	-.53396	-.28264
CGradientStatic	-.49992	-.36577	-.43563	-.63408
CHostnameSourceWnd	-.65407	-.33725	-.33909	-.64825
CHttpDownloadDlg	-.65407	-.49163	-.43563	-.77052
CHyperLink	-.35795	.29781	-.30871	-.17973
CHyperTextCtrl	.86130	-.02848	-.14422	.33553
CIconStatic	-.40994	-.21139	-.33909	-.46797
CInfoListCtrl	-.31566	-.13758	.83371	.18539
CIni	1.28921	.47430	-.53396	.59911
CInputBox	-.65407	-.36577	-.53396	-.75710
CIPFilter	-.28483	-.36577	-.53396	-.57719
CIrcMain	-.21750	-.18286	.05244	-.16953
CIrcSocket	-.02534	-.22315	2.15370	.92833
CIrcWnd	1.16837	-.12581	-.14243	.43859
CKeyWord	-.35795	.29781	-.30871	-.17973
CKnownFile	.48087	.36917	.24649	.53429
CKnownFileList	-.31180	1.29022	.30368	.62471
CLanCast	-.20702	-.33725	-.33909	-.43042
ClientsData	-.65407	-.49163	-.53396	-.81843
CLineInfo	1.17081	-.49163	-.53396	.07076
CLinePartInfo	1.24453	-.43459	-.14422	.32438
CListBoxST	1.36230	-.36577	-.53396	.22539

CListCtrlSorter	-.32053	.11745	-.14861	-.17136
CListCtrlSorterItem	.74280	.17194	-.30871	.29529
CListenSocket	-.16513	.20302	2.05098	1.01782
CLoggable	-.22390	-.49163	-.53396	-.60882
CMemDC	-.32139	.47430	-.53396	-.18567
CMemDC2	-.32139	-.49163	-.53396	-.65633
CMeterIcon	-.30953	-.36577	-.53396	-.58923
CMuleCtrlItem	-.35795	.53135	-.14422	.01422
CMuleListCtrl	.92257	-.23991	-.53396	.07245
CMuleRollup	-.43080	-.49163	-.53396	-.70964
CMuleSystrayDlg	-.23445	-.05201	-.43563	-.35185
CMuleToolbarCtrl	-.65407	.11745	-.24516	-.38093
CMyFont	-.16296	-.36577	-.53396	-.51781
CNewServerDlg	-.65407	-.33725	-.24076	-.60034
COptionTree	-.31253	2.32261	.05065	1.00411
COptionTreeCheckButton	-.40657	-.46311	-.33909	-.58898
COptionTreeColorPopUp	3.09593	-.49163	-.43563	1.05670
COptionTreeFileDlg	-.25792	-.36577	-.53396	-.56408
COptionTreeFontSel	-.51305	-.40606	.65232	-.12999
COptionTreeFontSelColorButton	-.45389	.60017	7.51239	3.73175
COptionTreeFontSelFontCombo	.08374	-.36577	-.33730	-.30178
COptionTreeFontSelSizeCombo	-.27264	-.36577	-.43563	-.52334
COptionTreeImagePopUp	.91353	-.46311	-.24076	.10216
COptionTreeInfo	-.46336	-.21139	-.33909	-.49400
COptionTreeIPAddressEdit	-.51068	-.36577	-.53396	-.68723
COptionTreeItem	.71666	2.63071	.73538	1.98936
COptionTreeItemCheckBox	.70097	-.32049	.93025	.63866
COptionTreeItemColor	.68836	-.34901	.73538	.52367
COptionTreeItemComboBox	.71022	-.34901	.73538	.53432
COptionTreeItemDate	.70975	-.34901	.73538	.53409
COptionTreeItemEdit	.69941	-.34901	.73538	.52905
COptionTreeItemFile	.70176	-.32049	.73538	.54410
COptionTreeItemFont	.71293	-.34901	.73538	.53564
COptionTreeItemHyperLink	.69455	-.34901	.73538	.52668
COptionTreeItemImage	.68816	-.34901	.73538	.52358
COptionTreeItemIPAddress	.69144	-.23492	.73538	.58077
COptionTreeItemRadio	.69224	-.32049	.93025	.63441
COptionTreeItemSpinner	-.22230	-.29197	.93025	.20269
COptionTreeItemStatic	.71845	-.34901	.73538	.53833
COptionTreeList	-.18822	-.21139	-.14243	-.26411
COptionTreeRadioButton	-.32858	-.46311	-.33909	-.55098
COptionTreeSpinnerButton	-.38468	-.30872	-.33909	-.50309
COptionTreeSpinnerEdit	.82574	-.21139	-.33909	.13413
COScopeCtrl	8.40274	-.36577	-.53396	3.65590
CPageSelectionBox	-.65407	-.49163	-.53396	-.81843
CPartFile	.35319	2.11881	.63801	1.51538
CPIDL	-.30060	-.49163	-.53396	-.64620

CPPgAdvanced	-.24573	-.33725	-.14243	-.35346
CPPgConnection	.01903	-.33725	-.14243	-.22445
CPPgDirectories	-.23684	-.28020	-.24076	-.36924
CPPgFiles	-.10426	-.33725	-.24076	-.33244
CPPgGeneral	-.25447	-.30872	-.24076	-.39173
CPPgHTTPD	-.47626	-.33725	-.24076	-.51370
CPPgIRC	-.23869	-.33725	-.24076	-.39794
CPPgMessaging	-.65407	-.33725	-.24076	-.60034
CPPgModPT	-.65407	-.33725	-.24076	-.60034
CPPgNotify	-.26858	-.33725	-.24076	-.41251
CPPgProxy	-.65407	-.33725	-.24076	-.60034
CPPgScheduler	-.31885	-.33725	-.24076	-.43700
CPPgServer	-.24510	-.33725	-.24076	-.40106
CPPgSorting	-.38422	-.08053	-.24076	-.34376
CPPgStats	-.29249	-.30872	-.24076	-.41026
CPPgWindow	-.65407	-.30872	-.14243	-.53853
CPPToolTip	.84907	-.28020	-.53396	.01701
CPreferences	-.36079	8.06950	-.23897	3.63969
CPreferencesDlg	-.65407	.14766	-.24076	-.36406
CPreparedHyperText	-.35581	.66948	-.15301	.07828
CPreviewThread	-.65407	-.46311	-.33909	-.70958
CProcessingCmdThread	.17043	-.36577	-.43563	-.30745
CProgressCtrlX	-.44583	-.36577	-.53396	-.65564
CQArray	-.29702	-.49163	-.43563	-.59654
CQueueListCtrl	-.25521	-.33725	-.23897	-.40512
CResizableDialog	-.65407	-.49163	-.53396	-.81843
CResizableFormView	-.65407	-.49163	-.53396	-.81843
CResizableFrame	-.65407	-.49163	-.53396	-.81843
CResizableGrip	-.65407	-.49163	-.53396	-.81843
CResizableLayout	-.65407	-.49163	-.53396	-.81843
CResizableMDIChild	-.65407	-.49163	-.53396	-.81843
CResizableMDIFrame	-.65407	-.49163	-.53396	-.81843
CResizableMinMax	-.65407	-.49163	-.53396	-.81843
CResizablePage	-.65407	-.49163	-.53396	-.81843
CResizablePageEx	-.65407	-.49163	-.53396	-.81843
CResizableSheet	.33569	-.49163	.65232	.24186
CResizableSheetEx	.33549	-.49163	-.53396	-.33626
CResizableState	-.65407	-.49163	-.53396	-.81843
CRollupCtrl	-.43080	-.36577	-.53396	-.64831
CRollupGripper	-.65407	.47430	-.53396	-.34777
CRollupHeader	-.45389	.47430	-.53396	-.25023
CRWLockLite	-.35795	-.46311	-.53396	-.66024
CSafeArray	-.29804	-.36577	-.53396	-.58363
CSafeArraySorted	-.29709	-.49163	-.53396	-.64449
CSafeFile	1.09032	-.49163	-.53396	.03154
CSafeMemFile	1.09032	-.49163	-.53396	.03154
CSaveDC	.38465	-.49163	-.53396	-.31231

CSearchDlg	-.48680	-.22315	.15255	-.27160
CSearchFile	-.35795	.04608	-.21038	-.25447
CSearchList	-.10779	.32633	.47613	.33848
CSearchListCtrl	.92014	.24331	.24470	.68613
CSecuredVar	-.22999	-.36577	-.53396	-.55047
CSelBitmap	-.41059	-.49163	-.53396	-.69979
CSelBkColor	-.41059	-.49163	-.53396	-.69979
CSelBkMode	.84920	-.49163	-.53396	-.08595
CSelBrush	-.41059	-.49163	-.53396	-.69979
CSelect	-.65407	-.49163	-.53396	-.81843
CSelFont	-.41059	-.46311	-.53396	-.68589
CSelMapMode	.84920	-.49163	-.53396	-.08595
CSelPalette	-.37947	-.49163	-.53396	-.68463
CSelPen	-.41059	-.49163	-.53396	-.69979
CSelROP2	.84920	-.49163	-.53396	-.08595
CSelStock	-.42278	-.49163	-.53396	-.70573
CSelTextAlign	-.41059	-.49163	-.53396	-.69979
CSelTextColor	-.41059	-.49163	-.53396	-.69979
CServer	-.30153	.85189	-.33730	.10381
CServerConnect	.25229	1.03530	.72480	.98055
CServerEntry	.12731	-.49163	-.53396	-.43770
CServerList	-.11701	1.04695	.40462	.65028
CServerListCtrl	-.47957	.11745	.04804	-.15304
CServerSocket	-.34703	1.44704	3.31753	2.15248
CServerWnd	-.45008	.68038	-.11205	.05762
CSharedFileList	-.28010	1.64172	1.31281	1.30314
CSharedFilesCtrl	-.06255	.72653	.63183	.63139
CSharedFilesWnd	-.20002	.23154	-.14683	-.05618
CShellContextMenu	-.22390	-.36577	-.43563	-.49959
CSourceEntry	-.16949	-.36577	-.53396	-.52099
CSplashScreen	.98024	-.43459	-.53396	.00570
CSplitterControl	-.34687	-.49163	-.53396	-.66875
CStatisticsData	-.02583	-.33725	-.53396	-.43709
CStatisticsDlg	-.26567	-.16610	1.04563	.29911
CStoredSources	-.15091	-.36577	-.53396	-.51194
CStoredSourcesContainer	-.21750	-.49163	-.53396	-.60571
CTag	-.11422	-.49163	-.43563	-.50747
CTaskbarNotifier	-.65407	-.36577	-.43563	-.70919
CThemeHelperST	-.24773	-.11405	-.53396	-.43646
CTitleMenu	-.28992	1.22948	-.53396	.19763
CTransferWnd	-.38766	-.19463	-.33730	-.44808
CTrayDialog	-.31406	-.49163	-.53396	-.65276
CTrayMenuBtn	.70132	-.36577	-.53396	-.09668
CUDPSocket	.91941	.38338	.17936	.72219
CUDPSocketWnd	-.65407	-.33725	-.33909	-.64825
CUpdateServerMetDlg	-.57695	.11745	-.05029	-.24840
CUpDownClient	.41827	4.80763	2.01597	3.52866

CUploadListCtrl	-.23979	-.33725	-.14065	-.34970
CUploadQueue	-.32140	.65516	2.64586	1.45184
CVisLine	-.65407	-.49163	-.53396	-.81843
CVisPart	1.39348	-.12581	.24553	.73732
CVisualStylesXP	-.24650	-.49163	-.53396	-.61984
CWebServer	-.46323	3.53722	1.33196	2.14684
CWebSocket	1.57572	.95752	-.14861	1.16193
CXPStyleButtonST	-.18187	-.33725	-.33909	-.41817
Db	.54302	1.43869	.45759	1.18857
Dbc	-.65407	.50078	-.11823	-.13230
DbDeadlockException	-.65407	-.49163	-.53396	-.81843
DbEnv	-.32139	4.43025	1.47512	2.72084
DbException	-.65407	-.49163	-.53396	-.81843
DbLock	-.65407	.62664	-.11823	-.07098
DbLockNotGrantedException	-.65407	-.43459	-.14422	-.60073
DbLogc	-.65407	.95547	.07224	.18206
DbLsn	-.65407	.58380	-.08346	-.07491
DbMemoryException	-.65407	-.46311	-.33909	-.70958
DbMpoolFile	.05728	.50078	-.11823	.21431
DbPreplist	-.05038	-.46311	-.33909	-.41542
DbRunRecoveryException	-.65407	-.49163	-.53396	-.81843
Dbt	-.35795	1.91095	.36705	.93556
DbTxn	-.32139	.17194	-.30871	-.22324
InputBox	-.65407	-.49163	-.43563	-.77052
MD5Sum	-.24827	-.49163	-.23897	-.47697
MiniDumper	-.40242	-.36577	-.33730	-.53866
Packet	2.49416	.13768	-.53396	1.02221
sfl_itemdata	.06096	-.46311	-.33909	-.36117
StatusBarCtrl	-.65407	-.43459	-.43563	-.74272
XBMDraw	.95522	-.49163	-.53396	-.03429
Zero	-.65407	-.49163	-.53396	-.81843

6. Appendix 6 - CPartFile Interface Dependence Data

6.1. Class CPartFile

The following tables list the measured data describing the levels of interface dependence present in the CPartFile class within the eMulePlus software system.

Class CPartFile

	interface method	CIEI1	CIEI2	CIEI3	CIEI4	CIEI5	CIEI6	CIEI7	CIEI8
1	AddClientSource	0	0	0	0	0	0	0	0
2	AddClientSources	0	0	0	0	0	0	4	0
3	AddClientSources	0	0	0	0	0	0	1	0
4	AddFromStoredSources	0	0	0	0	0	0	1	0
5	AddGap	0	0	0	0	0	0	1	0
6	AddSources	0	0	0	0	0	0	3	0
7	CharFillRange	0	0	0	0	0	0	0	0
8	CompleteFile	0	0	0	0	0	0	2	0
9	CreateED2KSourceLink	0	0	0	0	0	0	0	0
10	CreateFromFile	0	0	0	0	0	0	0	0
11	CreatePartFile	0	0	0	0	0	0	1	0
12	CreateSrcInfoPacket	0	0	0	0	0	0	1	0
13	DeleteFile	0	0	0	0	0	0	1	0
14	DrawStatusBar	0	0	0	0	0	0	1	0
15	FillGap	0	0	0	0	0	0	2	0
16	FirstLastLoaded	0	0	0	0	0	0	3	0
17	FlushBuffer	0	0	0	0	0	0	9	0
18	GetAvailablePartCount	0	0	0	0	0	0	0	0
19	GetCommonFilePenalty	0	0	0	0	0	0	0	0
20	GetCompletedSize	0	0	0	0	0	0	0	0
21	GetCompleteSourcesAccuracy	0	0	0	0	0	0	0	0
22	GetCompleteSourcesCount	0	0	0	0	0	0	0	0
23	GetDatarate	0	0	0	0	0	0	0	0
24	GetDiscardSuperCompressed	0	0	0	0	0	0	0	0
25	GetDownloadFileInfo	0	0	0	0	0	0	8	0
26	GetDownloadFileInfo4Tooltips	0	0	0	0	0	0	8	0
27	GetFilledList	0	0	0	0	0	0	0	0
28	GetFullName	0	0	0	0	0	0	0	0
29	GetGainDueToCompression	0	0	0	0	0	0	0	0
30	GetGapsInPart	0	0	0	0	0	0	1	0
31	GetLastAnsweredTime	0	0	0	0	0	0	0	0
32	GetLastDownTransfer	0	0	0	0	0	0	0	0
33	GetLoadedSourcesCompletely	0	0	0	0	0	0	0	0
34	GetLoadSourcesAtOnceLimit	0	0	0	0	0	0	0	0
35	GetLoadSourcesSlowTimeInterval	0	0	0	0	0	0	0	0
36	GetLoadSourcesTimeInterval	0	0	0	0	0	0	0	0
37	GetLostDueToCorruption	0	0	0	0	0	0	0	0
38	GetNextEmptyBlockInPart	0	0	0	0	0	0	1	0
39	GetNextRequestedBlock	0	0	0	0	0	0	5	0
40	GetNotCurrentSourcesCount	0	0	0	0	0	0	0	0
41	GetOutputDir	0	0	0	0	0	0	0	0
42	GetPartfileStatus	0	0	0	0	0	0	2	0
43	getPartfileStatusRang	0	0	0	0	0	0	2	0

44	GetPartMetFileName	0	0	0	0	0	0	0	0
45	GetPercentCompleted	0	0	0	0	0	0	0	0
46	GetPriority	0	0	0	0	0	0	0	0
47	GetProgressString	0	0	0	0	0	0	2	0
48	GetRating	0	0	0	0	0	0	0	0
49	GetRemainingBlocksInPart	0	0	0	0	0	0	2	0
50	GetSaveSourcesTimeInterval	0	0	0	0	0	0	0	0
51	GetSizeToTransferAndNeededSpace	0	0	0	0	0	0	0	0
52	GetSourceCount	0	0	0	0	0	0	0	0
53	GetSourcesAfterServerConnect	0	0	0	0	0	0	0	0
54	GetSrcpartFrequency	0	0	0	0	0	0	0	0
55	GetStatus	0	0	0	0	0	0	0	0
56	GetStoredSources	0	0	0	0	0	0	0	0
57	GetTempDir	0	0	0	0	0	0	0	0
58	GetTimeRemaining	0	0	0	0	0	0	2	0
59	GetTransfered	0	0	0	0	0	0	0	0
60	GetTransferingSrcCount	0	0	0	0	0	0	0	0
61	GetValidSourcesCount	0	0	0	0	0	0	0	0
62	HasComment	0	0	0	0	0	0	0	0
63	HashSinglePart	0	0	0	0	0	0	0	0
64	HasRating	0	0	0	0	0	0	0	0
65	Init	0	0	0	0	0	0	0	0
66	InitializeFromLink	0	0	0	0	0	0	2	0
67	IsA4AFAuto	0	0	0	0	0	0	0	0
68	IsAlreadyRequested	0	0	0	0	0	0	0	1
69	IsArchive	0	0	0	0	0	0	0	0
70	IsAutoPrioritized	0	0	0	0	0	0	0	0
71	IsAviMovie	0	0	0	0	0	0	0	0
72	IsBetterMovieChunk	0	0	0	0	0	0	1	0
73	IsComplete	0	0	0	0	0	0	0	0
74	IsCorruptedPart	0	0	0	0	0	0	0	0
75	IsMovie	0	0	0	0	0	0	2	0
76	IsMovieChunk	0	0	0	0	0	0	0	0
77	IsMpgMovie	0	0	0	0	0	0	1	0
78	IsPartFile	0	0	0	0	0	0	0	0
79	IsPureGap	0	0	0	0	0	0	0	0
80	IsVLCInstalled	0	0	0	0	0	0	0	0
81	LoadAndAddSources	0	0	0	0	0	0	4	0
82	LoadFromFile	0	0	0	0	0	0	0	0
83	LoadMovieMode	0	0	0	0	0	0	0	0
84	LoadPartFile	0	0	0	0	0	0	5	0
85	LoadSourcesFromFile	0	0	0	0	0	0	1	0
86	localelastdowntransfer	0	0	0	0	0	0	0	0
87	localelastseencomplete	0	0	0	0	0	0	0	0
88	Movie1	0	0	0	0	0	0	3	0
89	Movie2	0	0	0	0	0	0	3	0
90	NewSrcPartsInfo	0	0	0	0	0	0	1	0
91	PartFileHashFinished	0	0	0	0	0	0	4	0
92	PauseFile	0	0	0	0	0	0	2	0
93	PrepareComparePart	0	0	0	0	0	0	1	0
94	PreviewAvailable	0	0	0	0	0	0	6	0
95	PreviewFile	0	0	0	0	0	0	2	0
96	Process	0	0	0	0	0	0	11	0
97	RemoveAllRequestedBlocks	0	0	0	0	0	0	0	0
98	RemoveAllSources	0	0	0	0	0	0	1	0
99	RemoveBlockFromList	0	0	0	0	0	0	0	0
100	RemoveNoNeededSources	0	0	0	0	0	0	0	0

101	ResumeFile	0	0	0	0	0	0	3	0
102	SaveMovieMode	0	0	0	0	0	0	0	0
103	SavePartFile	0	0	0	0	0	0	1	0
104	SavePartFileStats	0	0	0	0	0	0	0	0
105	SaveSources	0	0	0	0	0	0	4	0
106	SaveSourcesToFile	0	0	0	0	0	0	2	0
107	SaveToStoredSources	0	0	0	0	0	0	1	0
108	SaveToStoredSources	0	0	0	0	0	0	1	0
109	SetA4AFAuto	0	0	0	0	0	0	0	0
110	SetAlternativeOutputDir	0	0	0	0	0	0	0	0
111	SetAutoPriority	0	0	0	0	0	0	0	0
112	SetDiscardSuperCompressed	0	0	0	0	0	0	0	0
113	SetHasComment	0	0	0	0	0	0	0	0
114	SetHasRating	0	0	0	0	0	0	0	0
115	SetLastAnsweredTime	0	0	0	0	0	0	0	0
116	SetLastAnsweredTimeTimeout	0	0	0	0	0	0	0	0
117	SetLoadedSourcesCompletely	0	0	0	0	0	0	0	0
118	SetLoadSourcesAtOnceLimit	0	0	0	0	0	0	0	0
119	SetLoadSourcesSlowTimeInterval	0	0	0	0	0	0	0	0
120	SetLoadSourcesTimeInterval	0	0	0	0	0	0	0	0
121	SetPriority	0	0	0	0	0	0	2	0
122	SetSaveSourcesTimeInterval	0	0	0	0	0	0	0	0
123	StopFile	0	0	0	0	0	0	4	0
124	TotalPacketsSavedDueToICH	0	0	0	0	0	0	0	0
125	UpdateAvailablePartsCount	0	0	0	0	0	0	0	0
126	UpdateCompletedInfos	0	0	0	0	0	0	0	0
127	UpdateDisplayedInfo	0	0	0	0	0	0	0	0
128	UpdateDownloadAutoPriority	0	0	0	0	0	0	2	0
129	UpdateFileRatingCommentAvail	0	0	0	0	0	0	1	0
130	WriteCompleteSourcesCount	0	0	0	0	0	0	1	0
131	WritePartStatus	0	0	0	0	0	0	1	0
132	WriteToBuffer	0	0	0	0	0	0	3	0
133	WriteToFile	0	0	0	0	0	0	0	0
Total	133	133	133	133	133	133	133	133	133

Class CPartFile

	class	CIS1	CIS2	CIS3	CIS4
1	CPartFile	0	34	133	5
Total	1	1	1	1	1

Class CPartFile

	interface method	CIS5	CIS6	CIS7	CIS8
1	AddClientSource	32	0	0	0
2	AddClientSources	25	4	1	0
3	AddClientSources	19	1	0	0
4	AddFromStoredSources	115	1	4	5
5	AddGap	35	1	0	0
6	AddSources	39	3	1	0
7	CharFillRange	6	0	0	0
8	CompleteFile	28	2	2	2
9	CreateED2KSourceLink	71	0	0	0
10	CreateFromFile	1	0	0	0
11	CreatePartFile	47	1	3	4
12	CreateSrcInfoPacket	74	1	0	0
13	DeleteFile	33	2	3	0
14	DrawStatusBar	116	1	5	4
15	FillGap	32	2	0	0
16	FirstLastLoaded	6	3	0	0
17	FlushBuffer	109	9	3	4
18	GetAvailablePartCount	1	0	0	0
19	GetCommonFilePenalty	3	0	0	0
20	GetCompletedSize	1	0	1	0
21	GetCompleteSourcesAccuracy	1	0	0	0
22	GetCompleteSourcesCount	17	0	1	1
23	GetDatarate	1	0	0	0
24	GetDiscardSuperCompressed	1	0	0	0
25	GetDownloadFileInfo	28	8	0	0
26	GetDownloadFileInfo4Tooltips	39	8	0	0
27	GetFilledList	43	0	0	0
28	GetFullName	1	0	0	0
29	GetGainDueToCompression	1	0	0	0
30	GetGapsInPart	36	1	0	0
31	GetLastAnsweredTime	1	0	0	0
32	GetLastDownTransfer	1	0	0	0
33	GetLoadedSourcesCompletely	1	0	0	0
34	GetLoadSourcesAtOnceLimit	1	0	0	0
35	GetLoadSourcesSlowTimeInterval	1	0	0	0
36	GetLoadSourcesTimeInterval	1	0	0	0
37	GetLostDueToCorruption	1	0	0	0
38	GetNextEmptyBlockInPart	57	1	0	0
39	GetNextRequestedBlock	81	5	0	0
40	GetNotCurrentSourcesCount	15	0	0	0
41	GetOutputDir	63	0	0	0
42	GetPartfileStatus	27	2	0	0
43	getPartfileStatusRang	28	2	0	0
44	GetPartMetFileName	1	0	0	0
45	GetPercentCompleted	1	0	0	0
46	GetPriority	1	0	0	0

47	GetProgressString	52	2	0	0
48	GetRating	26	0	0	0
49	GetRemainingBlocksInPart	20	2	0	0
50	GetSaveSourcesTimeInterval	1	0	0	0
51	GetSizeToTransferAndNeededSpace	11	0	0	0
52	GetSourceCount	8	0	0	0
53	GetSourcesAfterServerConnect	8	0	1	1
54	GetSrcpartFrequency	1	0	0	0
55	GetStatus	9	0	1	0
56	GetStoredSources	1	0	0	0
57	GetTempDir	1	0	0	0
58	GetTimeRemaining	8	2	0	0
59	GetTransfered	1	0	0	0
60	GetTransferingSrcCount	1	0	0	0
61	GetValidSourcesCount	16	0	0	0
62	HasComment	1	0	0	0
63	HashSinglePart	36	0	0	0
64	HasRating	1	0	0	0
65	Init	64	0	0	32
66	InitializeFromLink	29	2	0	1
67	IsA4AFAuto	1	0	0	0
68	IsAlreadyRequested	11	0	0	0
69	IsArchive	5	0	0	0
70	IsAutoPrioritized	1	0	0	0
71	IsAviMovie	9	0	0	0
72	IsBetterMovieChunk	35	1	0	0
73	IsComplete	17	0	0	0
74	IsCorruptedPart	7	0	0	0
75	IsMovie	4	2	0	0
76	IsMovieChunk	76	0	0	0
77	IsMpgMovie	14	1	0	0
78	IsPartFile	1	0	0	0
79	IsPureGap	14	0	0	0
80	IsVLCInstalled	12	0	0	0
81	LoadAndAddSources	49	4	4	0
82	LoadFromFile	1	0	0	0
83	LoadMovieMode	9	0	0	0
84	LoadPartFile	204	6	5	8
85	LoadSourcesFromFile	47	1	1	4
86	localelastdowntransfer	1	1	0	0
87	localelastseencomplete	1	1	0	0
88	Movie1	17	3	0	0
89	Movie2	25	3	0	0
90	NewSrcPartsInfo	27	1	0	0
91	PartFileHashFinished	36	4	1	1
92	PauseFile	24	2	1	2
93	PrepareComparePart	21	1	0	0
94	PreviewAvailable	43	6	1	0
95	PreviewFile	38	2	0	0
96	Process	229	11	13	9
97	RemoveAllRequestedBlocks	6	0	0	0
98	RemoveAllSources	17	1	0	0
99	RemoveBlockFromList	11	0	0	0
100	RemoveNoNeededSources	18	0	0	1
101	ResumeFile	13	3	1	1
102	SaveMovieMode	8	0	0	0
103	SavePartFile	126	1	5	0

104	SavePartFileStats	28	0	1	0
105	SaveSources	99	4	4	0
106	SaveSourcesToFile	50	2	2	1
107	SaveToStoredSources	45	1	4	0
108	SaveToStoredSources	40	1	4	0
109	SetA4AFAuto	13	0	0	0
110	SetAlternativeOutputDir	6	0	0	0
111	SetAutoPriority	1	0	0	0
112	SetDiscardSuperCompressed	1	0	0	0
113	SetHasComment	1	0	0	0
114	SetHasRating	1	0	0	0
115	SetLastAnsweredTime	1	0	0	0
116	SetLastAnsweredTimeTimeout	1	0	0	0
117	SetLoadedSourcesCompletely	1	0	0	0
118	SetLoadSourcesAtOnceLimit	1	0	0	0
119	SetLoadSourcesSlowTimeInterval	1	0	0	0
120	SetLoadSourcesTimeInterval	1	0	0	0
121	SetPriority	9	2	0	1
122	SetSaveSourcesTimeInterval	1	0	0	0
123	StopFile	13	4	0	2
124	TotalPacketsSavedDueToICH	1	0	0	0
125	UpdateAvailablePartsCount	24	0	1	1
126	UpdateCompletedInfos	18	0	0	2
127	UpdateDisplayedInfo	9	0	0	0
128	UpdateDownloadAutoPriority	17	2	1	0
129	UpdateFileRatingCommentAvail	18	1	0	0
130	WriteCompleteSourcesCount	5	1	0	0
131	WritePartStatus	19	1	0	0
132	WriteToBuffer	49	3	3	3
133	WriteToFile	1	0	0	0
Total	133	133	133	133	133

Class CPartFile

	class	CDE1
1	CPartFile	0
Total	1	1

Class CPartFile

	interface method	CDE2	CDE3
1	AddClientSource	0	0
2	AddClientSources	1	0
3	AddClientSources	0	0
4	AddFromStoredSources	4	5
5	AddGap	0	0
6	AddSources	1	0
7	CharFillRange	0	0
8	CompleteFile	2	2
9	CreateED2KSourceLink	0	0
10	CreateFromFile	0	0
11	CreatePartFile	3	4
12	CreateSrcInfoPacket	0	0
13	DeleteFile	3	0
14	DrawStatusBar	5	4
15	FillGap	0	0
16	FirstLastLoaded	0	0
17	FlushBuffer	3	4
18	GetAvailablePartCount	0	0
19	GetCommonFilePenalty	0	0
20	GetCompletedSize	1	0
21	GetCompleteSourcesAccuracy	0	0
22	GetCompleteSourcesCount	1	1
23	GetDatarate	0	0
24	GetDiscardSuperCompressed	0	0
25	GetDownloadFileInfo	0	0
26	GetDownloadFileInfo4Tooltips	0	0
27	GetFilledList	0	0
28	GetFullName	0	0
29	GetGainDueToCompression	0	0
30	GetGapsInPart	0	0
31	GetLastAnsweredTime	0	0
32	GetLastDownTransfer	0	0
33	GetLoadedSourcesCompletely	0	0
34	GetLoadSourcesAtOnceLimit	0	0
35	GetLoadSourcesSlowTimeInterval	0	0
36	GetLoadSourcesTimeInterval	0	0
37	GetLostDueToCorruption	0	0
38	GetNextEmptyBlockInPart	0	0
39	GetNextRequestedBlock	0	0
40	GetNotCurrentSourcesCount	0	0
41	GetOutputDir	0	0
42	GetPartfileStatus	0	0
43	getPartfileStatusRang	0	0
44	GetPartMetFileName	0	0
45	GetPercentCompleted	0	0
46	GetPriority	0	0
47	GetProgressString	0	0
48	GetRating	0	0
49	GetRemainingBlocksInPart	0	0
50	GetSaveSourcesTimeInterval	0	0
51	GetSizeToTransferAndNeededSpace	0	0
52	GetSourceCount	0	0
53	GetSourcesAfterServerConnect	1	1
54	GetSrcpartFrequency	0	0

55	GetStatus	1	0
56	GetStoredSources	0	0
57	GetTempDir	0	0
58	GetTimeRemaining	0	0
59	GetTransfered	0	0
60	GetTransferingSrcCount	0	0
61	GetValidSourcesCount	0	0
62	HasComment	0	0
63	HashSinglePart	0	0
64	HasRating	0	0
65	Init	0	32
66	InitializeFromLink	0	1
67	IsA4AFAuto	0	0
68	IsAlreadyRequested	0	0
69	IsArchive	0	0
70	IsAutoPrioritized	0	0
71	IsAviMovie	0	0
72	IsBetterMovieChunk	0	0
73	IsComplete	0	0
74	IsCorruptedPart	0	0
75	IsMovie	0	0
76	IsMovieChunk	0	0
77	IsMpgMovie	0	0
78	IsPartFile	0	0
79	IsPureGap	0	0
80	IsVLCInstalled	0	0
81	LoadAndAddSources	4	0
82	LoadFromFile	0	0
83	LoadMovieMode	0	0
84	LoadPartFile	5	8
85	LoadSourcesFromFile	1	4
86	localelastdowntransfer	0	0
87	localelastseencomplete	0	0
88	Movie1	0	0
89	Movie2	0	0
90	NewSrcPartsInfo	0	0
91	PartFileHashFinished	1	1
92	PauseFile	1	2
93	PrepareComparePart	0	0
94	PreviewAvailable	1	0
95	PreviewFile	0	0
96	Process	13	9
97	RemoveAllRequestedBlocks	0	0
98	RemoveAllSources	0	0
99	RemoveBlockFromList	0	0
100	RemoveNoNeededSources	0	1
101	ResumeFile	1	1
102	SaveMovieMode	0	0
103	SavePartFile	5	0
104	SavePartFileStats	1	0
105	SaveSources	4	0
106	SaveSourcesToFile	2	1
107	SaveToStoredSources	4	0
108	SaveToStoredSources	4	0
109	SetA4AFAuto	0	0
110	SetAlternativeOutputDir	0	0
111	SetAutoPriority	0	0

112	SetDiscardSuperCompressed	0	0
113	SetHasComment	0	0
114	SetHasRating	0	0
115	SetLastAnsweredTime	0	0
116	SetLastAnsweredTimeTimeout	0	0
117	SetLoadedSourcesCompletely	0	0
118	SetLoadSourcesAtOnceLimit	0	0
119	SetLoadSourcesSlowTimeInterval	0	0
120	SetLoadSourcesTimeInterval	0	0
121	SetPriority	0	1
122	SetSaveSourcesTimeInterval	0	0
123	StopFile	0	2
124	TotalPacketsSavedDueToICH	0	0
125	UpdateAvailablePartsCount	1	1
126	UpdateCompletedInfos	0	2
127	UpdateDisplayedInfo	0	0
128	UpdateDownloadAutoPriority	1	0
129	UpdateFileRatingCommentAvail	0	0
130	WriteCompleteSourcesCount	0	0
131	WritePartStatus	0	0
132	WriteToBuffer	3	3
133	WriteToFile	0	0
Total	133	133	133

Class CPartFile

	member attribute	protection	CDE4	CDE5
1	availablePartsCount	private	2	1
2	completedsize	private	3	3
3	confirmedsize	private	1	1
4	count	private	2	1
5	datarate	private	5	1
6	fullname	private	3	5
7	lastpurgetime	private	3	3
8	lastsearchtime	private	4	2
9	m_ClientSrcAnswered	private	1	0
10	m_iGainDueToCompression	private	3	1
11	m_iLastTimeSourcesLoaded	private	2	2
12	m_iLastTimeSourcesLoadedPartial	private	3	2
13	m_iLastTimeSourcesSaved	private	3	2
14	m_iLoadingSourcesPartialTimeInterval	private	2	5
15	m_iLoadingSourcesTimeInterval	private	3	4
16	m_iLoadSourcesAtOnceLimit	private	2	4
17	m_iLostDueToCorruption	private	2	0
18	m_iSavingSourcesTimeInterval	private	2	4
19	m_iSourceIDIndex	private	2	1
20	m_iSourceIndex	private	2	1
21	m_iTotalPacketsSavedDueToICH	private	2	1
22	m_LastNoNeededCheck	private	2	1
23	m_nLastBufferFlushTime	private	2	1
24	m_nLastCompleteSrcCount	private	2	1
25	m_nSavedReduceDownload	private	2	1
26	m_nTotalBufferData	private	3	3
27	partmetfilename	private	3	5
28	percentcompleted	private	3	1

29	percentconfirmed	private	1	1
30	priority	private	3	3
31	status	private	7	7
32	tempdir	private	3	6
33	transfered	private	3	3
34	transferingsrc	private	4	1
Total	34	34	34	34

6.2. Object CPartFile

The following tables list the measured data describing the levels of interface dependence present in the CPartFile object within the eMulePlus software system.

Object CPartFile

	interface method	OIEI1	OIEI2	OIEI3	OIEI4	OIEI5	OIEI6	OIEI7	OIEI8
1	AddClientSource	0	0	0	0	0	0	0	0
2	AddClientSources	0	0	0	0	0	0	4	0
3	AddClientSources	0	0	0	0	0	0	1	0
4	AddDebugLogLine	0	0	0	0	0	0	0	0
5	AddFromStoredSources	0	0	0	0	0	0	3	0
6	AddGap	0	0	0	0	0	0	1	0
7	AddLogLine	0	0	0	0	0	0	0	0
8	AddSources	0	0	0	0	0	0	3	0
9	CalculateCompleteSources	0	0	0	0	0	0	0	0
10	CharFillRange	0	0	0	0	0	0	0	0
11	CreateED2KSourceLink	0	0	0	0	0	0	5	0
12	CreateFromFile	0	1	0	0	0	0	0	0
13	CreateFromFile	0	0	0	0	0	0	0	0
14	CreateSrcInfoPacket	0	0	0	0	0	0	2	0
15	CreateSrcInfoPacket	0	0	0	0	0	0	0	0
16	DeleteFile	0	0	0	2	1	1	2	21
17	DrawStatusBar	0	0	0	0	0	0	2	0
18	FillGap	0	0	0	0	0	0	2	0
19	FirstLastLoaded	0	0	0	0	0	0	4	2
20	FlushBuffer	0	0	0	2	1	1	10	21
21	GetAvailablePartCount	0	0	0	0	0	0	0	0
22	GetBlockCount	0	0	0	0	0	0	0	0
23	GetBlockSize	0	0	0	0	0	0	1	0
24	GetBlockTraffic	0	0	0	0	0	0	0	0
25	GetCommonFilePenalty	0	0	0	0	0	0	0	0
26	GetCompletedSize	0	0	0	0	0	0	0	0
27	GetCompleteSourcesAccuracy	0	0	0	0	0	0	0	0
28	GetCompleteSourcesAccuracy	0	0	0	0	0	0	0	0
29	GetCompleteSourcesCount	0	0	0	0	0	0	0	0
30	GetCompleteSourcesCount	0	0	0	0	0	0	0	0
31	GetDatarate	0	0	0	0	0	0	0	0
32	GetDiscardSuperCompressed	0	0	0	0	0	0	0	0
33	GetDownloadFileInfo	0	0	0	0	0	0	13	2
34	GetDownloadFileInfo4Tooltips	0	0	0	0	0	0	13	2
35	GetFileComment	0	0	0	0	0	0	0	0
36	GetFileDate	0	0	0	0	0	0	0	0
37	GetFileHash	0	0	0	0	0	0	0	0
38	GetFileName	0	0	0	0	0	0	0	0
39	GetFileRate	0	0	0	0	0	0	0	0
40	GetFileRatio	0	0	0	0	0	0	0	0
41	GetFileSize	0	0	0	0	0	0	0	0
42	GetFileType	0	0	0	0	0	0	0	0
43	GetFilledList	0	0	0	0	0	0	0	0
44	GetFullName	0	0	0	0	0	0	0	0
45	GetGainDueToCompression	0	0	0	0	0	0	0	0
46	GetGapsInPart	0	0	0	0	0	0	2	0

47	GetHashCount	0	0	0	0	0	0	0	0
48	GetLastAnsweredTime	0	0	0	0	0	0	0	0
49	GetLastDownTransfer	0	0	0	0	0	0	0	0
50	GetLoadedSourcesCompletely	0	0	0	0	0	0	0	0
51	GetLoadSourcesAtOnceLimit	0	0	0	0	0	0	0	0
52	GetLoadSourcesSlowTimeInterval	0	0	0	0	0	0	0	0
53	GetLoadSourcesTimeInterval	0	0	0	0	0	0	0	0
54	GetLostDueToCorruption	0	0	0	0	0	0	0	0
55	GetMovieMode	1	0	0	0	0	0	0	0
56	GetNextRequestedBlock	0	0	0	1	0	0	6	12
57	GetNotCurrentSourcesCount	0	0	0	0	0	0	0	0
58	GetOutputDir	0	0	0	0	0	0	0	0
59	GetPartCount	0	0	0	0	0	0	0	0
60	GetPartfileStatus	0	0	0	0	0	0	2	0
61	getPartfileStatusRang	0	0	0	0	0	0	2	0
62	GetPartHash	0	0	0	0	0	0	0	0
63	GetPartMetFileName	0	0	0	0	0	0	0	0
64	GetPartSize	0	0	0	0	0	0	1	0
65	GetPartStatus	0	0	0	0	0	0	0	0
66	GetPartTraffic	0	0	0	0	0	0	0	0
67	GetPath	0	0	0	0	0	0	0	0
68	GetPercentCompleted	0	0	0	0	0	0	0	0
69	GetPermissions	1	0	0	0	0	0	0	0
70	GetPriority	1	0	0	0	0	0	0	0
71	GetPriority	0	0	0	0	0	0	0	0
72	GetProgressString	0	0	0	0	0	0	4	0
73	GetRating	0	0	0	0	0	0	0	0
74	GetRemainingBlocksInPart	0	0	0	0	0	0	2	0
75	GetSaveSourcesTimeInterval	0	0	0	0	0	0	0	0
76	GetSharedFile	0	0	0	0	0	0	0	0
77	GetSizeToTransferAndNeededSpace	0	0	0	0	0	0	1	0
78	GetSourceCount	0	0	0	0	0	0	0	0
79	GetSourcesAfterServerConnect	0	0	0	0	0	0	0	0
80	GetSrcpartFrequency	0	0	0	0	0	0	0	0
81	GetStatus	0	0	0	0	0	0	0	0
82	GetStoredSources	0	0	0	0	0	0	0	0
83	GetTempDir	0	0	0	0	0	0	0	0
84	GetTimeRemaining	0	0	0	0	0	0	3	0
85	GetTrafficBlock	0	0	0	0	0	0	1	1
86	GetTrafficPart	0	0	0	0	0	0	1	1
87	GetTransfered	0	0	0	0	0	0	0	0
88	GetTransferingSrcCount	0	0	0	0	0	0	0	0
89	GetValidSourcesCount	0	0	0	0	0	0	0	0
90	HasComment	0	0	0	0	0	0	0	0
91	HasHiddenParts	0	0	0	0	0	0	0	0
92	HashSinglePart	0	0	0	0	0	0	5	0
93	HasRating	0	0	0	0	0	0	0	0
94	InitializeFromLink	0	0	0	2	2	2	1	9
95	IsA4AFAuto	0	0	0	0	0	0	0	0
96	IsArchive	0	0	0	0	0	0	1	0
97	IsAutoPrioritized	0	0	0	0	0	0	0	0
98	IsAutoPrioritized	0	0	0	0	0	0	0	0
99	IsAviMovie	0	0	0	0	0	0	1	0
100	IsBetterMovieChunk	0	0	0	0	0	0	3	0
101	IsComplete	0	0	0	0	0	0	0	0
102	IsCorruptedPart	0	0	0	0	0	0	0	0
103	IsMovie	0	0	0	0	0	0	2	2

104	IsMovieChunk	0	0	0	0	0	0	2	0
105	IsMpgMovie	0	0	0	0	0	0	2	0
106	IsPartFile	0	0	0	0	0	0	0	0
107	IsPartFile	0	0	0	0	0	0	0	0
108	IsPureGap	0	0	0	0	0	0	0	0
109	IsVLCInstalled	0	0	0	0	0	0	0	0
110	LoadAndAddSources	0	0	0	0	0	0	6	5
111	LoadFromFile	0	0	0	2	2	1	3	7
112	LoadFromFile	0	0	0	0	0	0	0	0
113	LoadFromFileTraffic	0	0	0	0	0	0	3	0
114	LoadHashsetFromFile	0	0	0	0	0	0	2	0
115	LoadMovieMode	0	0	0	0	1	0	1	0
116	LoadPartFile	1	0	0	2	3	2	11	27
117	LoadSourcesFromFile	0	0	0	0	0	0	4	0
118	localelastdowntransfer	0	0	0	0	0	0	0	0
119	localelastseencomplete	0	0	0	0	0	0	0	0
120	Movie1	0	0	0	0	0	0	3	7
121	Movie2	0	0	0	0	0	0	3	7
122	NewSrcPartsInfo	0	0	0	0	0	0	2	0
123	PartFileHashFinished	0	0	0	2	1	1	4	21
124	PauseFile	0	0	0	2	1	1	2	3
125	PrepareComparePart	0	0	0	0	0	0	3	2
126	PreviewAvailable	0	0	0	0	0	0	8	4
127	PreviewFile	0	0	0	0	0	0	2	10
128	Process	0	0	0	2	1	1	13	28
129	RemoveAllRequestedBlocks	0	0	0	0	0	0	0	0
130	RemoveAllSources	0	0	0	0	0	0	1	1
131	RemoveBlockFromList	0	0	0	0	0	0	0	0
132	RemoveNoNeededSources	0	0	0	0	0	0	0	0
133	ResumeFile	0	0	0	2	1	1	3	3
134	SaveMovieMode	0	0	0	0	0	0	0	0
135	SavePartFile	2	1	1	0	0	0	3	0
136	SavePartFileStats	0	0	0	0	0	0	1	2
137	SaveSources	0	0	0	0	0	0	6	5
138	SaveSourcesToFile	0	0	0	0	0	0	4	0
139	SaveToFileTraffic	0	0	0	0	0	0	2	0
140	SaveToStoredSources	0	0	0	0	0	0	3	0
141	SaveToStoredSources	0	0	0	0	0	0	3	0
142	SetA4AFAuto	0	0	0	0	0	0	0	0
143	SetAlternativeOutputDir	0	0	0	0	0	0	0	0
144	SetAutoPriority	0	0	0	0	0	0	0	0
145	SetAutoPriority	0	0	0	0	0	0	0	0
146	SetDiscardSuperCompressed	0	0	0	0	0	0	0	0
147	SetFileComment	0	0	0	0	0	0	0	0
148	SetFileName	0	0	0	0	0	0	0	0
149	SetFileRate	0	0	0	0	0	0	0	0
150	SetHasComment	0	0	0	0	0	0	0	0
151	SetHasRating	0	0	0	0	0	0	0	0
152	SetLastAnsweredTime	0	0	0	0	0	0	0	0
153	SetLastAnsweredTimeTimeout	0	0	0	0	0	0	0	0
154	SetLoadedSourcesCompletely	0	0	0	0	0	0	0	0
155	SetLoadSourcesAtOnceLimit	0	0	0	0	0	0	0	0
156	SetLoadSourcesSlowTimeInterval	0	0	0	0	0	0	0	0
157	SetLoadSourcesTimeInterval	0	0	0	0	0	0	0	0
158	SetMovieMode	0	1	0	0	0	0	0	0
159	SetPartStatus	0	0	0	0	0	0	0	0
160	SetPath	0	0	0	0	0	0	0	0

161	SetPermissions	0	1	0	0	0	0	0	0
162	SetPriority	0	0	0	2	1	1	2	3
163	SetPriority	0	1	0	0	0	0	0	0
164	SetSaveSourcesTimeInterval	0	0	0	0	0	0	0	0
165	SetSharedFile	0	0	0	0	0	0	0	0
166	StopFile	0	0	0	2	1	1	4	21
167	TotalPacketsSavedDueToICH	0	0	0	0	0	0	0	0
168	UpdateAvailablePartsCount	0	0	0	0	0	0	1	0
169	UpdateCompletedInfos	0	0	0	0	0	0	0	0
170	UpdateDisplayedInfo	0	0	0	0	0	0	0	0
171	UpdateDownloadAutoPriority	0	0	0	2	1	1	3	5
172	UpdateFileRatingCommentAvail	0	0	0	0	0	0	1	0
173	UpdateUploadAutoPriority	0	0	0	0	1	0	3	0
174	WriteCompleteSourcesCount	0	0	0	0	0	0	1	0
175	WritePartStatus	0	0	0	0	0	0	3	0
176	WritePartStatus	0	0	0	0	0	0	1	0
177	WriteToBuffer	0	0	0	2	1	1	5	21
178	WriteToFile	3	0	0	0	0	0	2	0
179	WriteToFile	0	0	0	0	0	0	0	0
Total	179	179	179	179	179	179	179	179	179

Object CPartFile

	object	OIS1	OIS2	OIS3	OIS4
1	CPartFile	4	43	179	16
Total	1	1	1	1	1

Object CPartFile

	interface method	OIS5	OIS6	OIS7	OIS8
1	AddClientSource	32	0	0	0
2	AddClientSources	25	4	1	0
3	AddClientSources	19	1	0	0
4	AddDebugLogLine	13	0	0	0
5	AddFromStoredSources	115	3	4	5
6	AddGap	35	1	0	0
7	AddLogLine	11	0	0	0
8	AddSources	39	3	1	0
9	CalculateCompleteSources	39	0	1	3
10	CharFillRange	6	0	0	0
11	CreateED2KSourceLink	71	5	0	0
12	CreateFromFile	48	2	2	4
13	CreateFromFile	1	0	0	0
14	CreateSrcInfoPacket	74	2	1	0
15	CreateSrcInfoPacket	44	0	0	0
16	DeleteFile	33	3	3	0
17	DrawStatusBar	116	2	6	4
18	FillGap	32	2	0	0
19	FirstLastLoaded	6	4	0	0
20	FlushBuffer	109	11	4	4
21	GetAvailablePartCount	1	0	0	0
22	GetBlockCount	7	0	1	0
23	GetBlockSize	9	1	1	0
24	GetBlockTraffic	11	0	0	0
25	GetCommonFilePenalty	3	0	0	0

26	GetCompletedSize	1	0	1	0
27	GetCompleteSourcesAccuracy	1	0	0	0
28	GetCompleteSourcesAccuracy	1	0	0	0
29	GetCompleteSourcesCount	17	0	1	1
30	GetCompleteSourcesCount	1	0	0	0
31	GetDatarate	1	0	0	0
32	GetDiscardSuperCompressed	1	0	0	0
33	GetDownloadFileInfo	28	13	0	0
34	GetDownloadFileInfo4Tooltips	39	13	0	0
35	GetFileComment	1	0	0	0
36	GetFileDate	1	0	0	0
37	GetFileHash	1	0	0	0
38	GetFileName	1	0	0	0
39	GetFileRate	1	0	1	0
40	GetFileRatio	17	0	0	0
41	GetFileSize	1	0	0	0
42	GetFileType	1	0	0	0
43	GetFilledList	43	0	1	0
44	GetFullName	1	0	0	0
45	GetGainDueToCompression	1	0	0	0
46	GetGapsInPart	36	3	0	0
47	GetHashCount	1	0	0	0
48	GetLastAnsweredTime	1	0	0	0
49	GetLastDownTransfer	1	0	0	0
50	GetLoadedSourcesCompletely	1	0	0	0
51	GetLoadSourcesAtOnceLimit	1	0	0	0
52	GetLoadSourcesSlowTimeInterval	1	0	0	0
53	GetLoadSourcesTimeInterval	1	0	0	0
54	GetLostDueToCorruption	1	0	0	0
55	GetMovieMode	1	0	1	0
56	GetNextRequestedBlock	81	7	0	0
57	GetNotCurrentSourcesCount	15	0	0	0
58	GetOutputDir	63	0	0	0
59	GetPartCount	7	0	1	0
60	GetPartfileStatus	27	2	0	0
61	getPartfileStatusRang	28	2	0	0
62	GetPartHash	9	0	0	0
63	GetPartMetFileName	1	0	0	0
64	GetPartSize	9	1	1	0
65	GetPartStatus	9	0	0	0
66	GetPartTraffic	11	0	0	0
67	GetPath	1	0	0	0
68	GetPercentCompleted	1	0	0	0
69	GetPermissions	1	0	1	0
70	GetPriority	1	0	1	0
71	GetPriority	1	0	0	0
72	GetProgressString	52	4	0	0
73	GetRating	26	0	0	0
74	GetRemainingBlocksInPart	20	3	0	0
75	GetSaveSourcesTimeInterval	1	0	0	0
76	GetSharedFile	1	0	0	0
77	GetSizeToTransferAndNeededSpace	11	1	0	0
78	GetSourceCount	8	0	0	0
79	GetSourcesAfterServerConnect	8	0	1	1
80	GetSrcpartFrequency	1	0	0	0
81	GetStatus	9	0	1	0
82	GetStoredSources	1	0	0	0

83	GetTempDir	1	0	0	0
84	GetTimeRemaining	8	3	0	0
85	GetTrafficBlock	34	1	0	0
86	GetTrafficPart	34	1	0	0
87	GetTransfered	1	0	0	0
88	GetTransferingSrcCount	1	0	0	0
89	GetValidSourcesCount	16	0	0	0
90	HasComment	1	0	0	0
91	HasHiddenParts	15	0	0	0
92	HashSinglePart	36	6	1	0
93	HasRating	1	0	0	0
94	InitializeFromLink	29	3	3	3
95	IsA4AFAuto	1	0	0	0
96	IsArchive	5	1	0	0
97	IsAutoPrioritized	1	0	0	0
98	IsAutoPrioritized	1	0	0	0
99	IsAviMovie	9	1	0	0
100	IsBetterMovieChunk	35	3	0	0
101	IsComplete	17	0	1	0
102	IsCorruptedPart	7	0	0	0
103	IsMovie	4	2	0	0
104	IsMovieChunk	76	2	0	0
105	IsMpgMovie	14	2	0	0
106	IsPartFile	1	0	0	0
107	IsPartFile	1	0	0	0
108	IsPureGap	14	0	1	0
109	IsVLCInstalled	12	0	0	0
110	LoadAndAddSources	49	6	4	0
111	LoadFromFile	26	5	1	0
112	LoadFromFile	1	0	0	0
113	LoadFromFileTraffic	29	3	0	0
114	LoadHashsetFromFile	62	3	2	0
115	LoadMovieMode	9	1	0	0
116	LoadPartFile	204	14	8	11
117	LoadSourcesFromFile	47	4	1	4
118	localelastdowntransfer	1	1	0	0
119	localelastseencomplete	1	1	0	0
120	Movie1	17	3	0	0
121	Movie2	25	3	0	0
122	NewSrcPartsInfo	27	2	0	0
123	PartFileHashFinished	36	5	3	1
124	PauseFile	24	2	1	2
125	PrepareComparePart	21	3	0	0
126	PreviewAvailable	43	8	1	0
127	PreviewFile	38	2	0	0
128	Process	229	13	14	9
129	RemoveAllRequestedBlocks	6	0	0	0
130	RemoveAllSources	17	1	0	0
131	RemoveBlockFromList	11	0	0	0
132	RemoveNoNeededSources	18	0	0	1
133	ResumeFile	13	3	1	1
134	SaveMovieMode	8	0	0	0
135	SavePartFile	126	3	10	1
136	SavePartFileStats	28	1	1	0
137	SaveSources	99	6	4	0
138	SaveSourcesToFile	50	4	2	1
139	SaveToFileTraffic	31	2	0	0

140	SaveToStoredSources	45	3	4	0
141	SaveToStoredSources	40	3	4	0
142	SetA4AFAuto	13	0	0	0
143	SetAlternativeOutputDir	6	0	0	0
144	SetAutoPriority	1	0	0	0
145	SetAutoPriority	1	0	0	0
146	SetDiscardSuperCompressed	1	0	0	0
147	SetFileComment	21	0	1	0
148	SetFileName	22	0	1	1
149	SetFileRate	21	0	1	1
150	SetHasComment	1	0	0	0
151	SetHasRating	1	0	0	0
152	SetLastAnsweredTime	1	0	0	0
153	SetLastAnsweredTimeTimeout	1	0	0	0
154	SetLoadedSourcesCompletely	1	0	0	0
155	SetLoadSourcesAtOnceLimit	1	0	0	0
156	SetLoadSourcesSlowTimeInterval	1	0	0	0
157	SetLoadSourcesTimeInterval	1	0	0	0
158	SetMovieMode	1	0	0	1
159	SetPartStatus	1	0	0	0
160	SetPath	8	0	1	1
161	SetPermissions	1	0	0	1
162	SetPriority	9	2	0	1
163	SetPriority	1	0	0	1
164	SetSaveSourcesTimeInterval	1	0	0	0
165	SetSharedFile	1	0	0	0
166	StopFile	13	4	0	2
167	TotalPacketsSavedDueToICH	1	0	0	0
168	UpdateAvailablePartsCount	24	1	1	1
169	UpdateCompletedInfos	18	0	1	2
170	UpdateDisplayedInfo	9	0	0	0
171	UpdateDownloadAutoPriority	17	3	1	0
172	UpdateFileRatingCommentAvail	18	1	0	0
173	UpdateUploadAutoPriority	18	3	0	0
174	WriteCompleteSourcesCount	5	1	0	0
175	WritePartStatus	20	1	0	0
176	WritePartStatus	19	3	0	0
177	WriteToBuffer	49	5	3	3
178	WriteToFile	47	2	6	0
179	WriteToFile	1	0	0	0
Total	179	179	179	179	179

Object CPartFile

	object	ODE1
1	CPartFile	4
Total	1	1

Object CPartFile

	interface method	ODE2	ODE3
1	AddClientSource	0	0
2	AddClientSources	1	0
3	AddClientSources	0	0
4	AddDebugLogLine	0	0

5	AddFromStoredSources	4	5
6	AddGap	0	0
7	AddLogLine	0	0
8	AddSources	1	0
9	CalculateCompleteSources	1	3
10	CharFillRange	0	0
11	CreateED2KSourceLink	0	0
12	CreateFromFile	2	4
13	CreateFromFile	0	0
14	CreateSrcInfoPacket	1	0
15	CreateSrcInfoPacket	0	0
16	DeleteFile	3	0
17	DrawStatusBar	6	4
18	FillGap	0	0
19	FirstLastLoaded	0	0
20	FlushBuffer	4	4
21	GetAvailablePartCount	0	0
22	GetBlockCount	1	0
23	GetBlockSize	1	0
24	GetBlockTraffic	0	0
25	GetCommonFilePenalty	0	0
26	GetCompletedSize	1	0
27	GetCompleteSourcesAccuracy	0	0
28	GetCompleteSourcesAccuracy	0	0
29	GetCompleteSourcesCount	1	1
30	GetCompleteSourcesCount	0	0
31	GetDatarate	0	0
32	GetDiscardSuperCompressed	0	0
33	GetDownloadFileInfo	0	0
34	GetDownloadFileInfo4Tooltips	0	0
35	GetFileComment	0	0
36	GetFileDate	0	0
37	GetFileHash	0	0
38	GetFileName	0	0
39	GetFileRate	1	0
40	GetFileRatio	0	0
41	GetFileSize	0	0
42	GetFileType	0	0
43	GetFilledList	1	0
44	GetFullName	0	0
45	GetGainDueToCompression	0	0
46	GetGapsInPart	0	0
47	GetHashCount	0	0
48	GetLastAnsweredTime	0	0
49	GetLastDownTransfer	0	0
50	GetLoadedSourcesCompletely	0	0
51	GetLoadSourcesAtOnceLimit	0	0
52	GetLoadSourcesSlowTimeInterval	0	0
53	GetLoadSourcesTimeInterval	0	0
54	GetLostDueToCorruption	0	0
55	GetMovieMode	1	0
56	GetNextRequestedBlock	0	0
57	GetNotCurrentSourcesCount	0	0
58	GetOutputDir	0	0
59	GetPartCount	1	0
60	GetPartfileStatus	0	0
61	getPartfileStatusRang	0	0

62	GetPartHash	0	0
63	GetPartMetFileName	0	0
64	GetPartSize	1	0
65	GetPartStatus	0	0
66	GetPartTraffic	0	0
67	GetPath	0	0
68	GetPercentCompleted	0	0
69	GetPermissions	1	0
70	GetPriority	1	0
71	GetPriority	0	0
72	GetProgressString	0	0
73	GetRating	0	0
74	GetRemainingBlocksInPart	0	0
75	GetSaveSourcesTimeInterval	0	0
76	GetSharedFile	0	0
77	GetSizeToTransferAndNeededSpace	0	0
78	GetSourceCount	0	0
79	GetSourcesAfterServerConnect	1	1
80	GetSrcpartFrequency	0	0
81	GetStatus	1	0
82	GetStoredSources	0	0
83	GetTempDir	0	0
84	GetTimeRemaining	0	0
85	GetTrafficBlock	0	0
86	GetTrafficPart	0	0
87	GetTransfered	0	0
88	GetTransferingSrcCount	0	0
89	GetValidSourcesCount	0	0
90	HasComment	0	0
91	HasHiddenParts	0	0
92	HashSinglePart	1	0
93	HasRating	0	0
94	InitializeFromLink	3	3
95	IsA4AFAuto	0	0
96	IsArchive	0	0
97	IsAutoPrioritized	0	0
98	IsAutoPrioritized	0	0
99	IsAviMovie	0	0
100	IsBetterMovieChunk	0	0
101	IsComplete	1	0
102	IsCorruptedPart	0	0
103	IsMovie	0	0
104	IsMovieChunk	0	0
105	IsMpgMovie	0	0
106	IsPartFile	0	0
107	IsPartFile	0	0
108	IsPureGap	1	0
109	IsVLCInstalled	0	0
110	LoadAndAddSources	4	0
111	LoadFromFile	1	0
112	LoadFromFile	0	0
113	LoadFromFileTraffic	0	0
114	LoadHashsetFromFile	2	0
115	LoadMovieMode	0	0
116	LoadPartFile	8	11
117	LoadSourcesFromFile	1	4
118	localelastdowntransfer	0	0

119	localelastseencomplete	0	0
120	Movie1	0	0
121	Movie2	0	0
122	NewSrcPartsInfo	0	0
123	PartFileHashFinished	3	1
124	PauseFile	1	2
125	PrepareComparePart	0	0
126	PreviewAvailable	1	0
127	PreviewFile	0	0
128	Process	14	9
129	RemoveAllRequestedBlocks	0	0
130	RemoveAllSources	0	0
131	RemoveBlockFromList	0	0
132	RemoveNoNeededSources	0	1
133	ResumeFile	1	1
134	SaveMovieMode	0	0
135	SavePartFile	10	1
136	SavePartFileStats	1	0
137	SaveSources	4	0
138	SaveSourcesToFile	2	1
139	SaveToFileTraffic	0	0
140	SaveToStoredSources	4	0
141	SaveToStoredSources	4	0
142	SetA4AFAuto	0	0
143	SetAlternativeOutputDir	0	0
144	SetAutoPriority	0	0
145	SetAutoPriority	0	0
146	SetDiscardSuperCompressed	0	0
147	SetFileComment	1	0
148	SetFileName	1	1
149	SetFileRate	1	1
150	SetHasComment	0	0
151	SetHasRating	0	0
152	SetLastAnsweredTime	0	0
153	SetLastAnsweredTimeTimeout	0	0
154	SetLoadedSourcesCompletely	0	0
155	SetLoadSourcesAtOnceLimit	0	0
156	SetLoadSourcesSlowTimeInterval	0	0
157	SetLoadSourcesTimeInterval	0	0
158	SetMovieMode	0	1
159	SetPartStatus	0	0
160	SetPath	1	1
161	SetPermissions	0	1
162	SetPriority	0	1
163	SetPriority	0	1
164	SetSaveSourcesTimeInterval	0	0
165	SetSharedFile	0	0
166	StopFile	0	2
167	TotalPacketsSavedDueToICH	0	0
168	UpdateAvailablePartsCount	1	1
169	UpdateCompletedInfos	1	2
170	UpdateDisplayedInfo	0	0
171	UpdateDownloadAutoPriority	1	0
172	UpdateFileRatingCommentAvail	0	0
173	UpdateUploadAutoPriority	0	0
174	WriteCompleteSourcesCount	0	0
175	WritePartStatus	0	0

176	WritePartStatus	0	0
177	WriteToBuffer	3	3
178	WriteToFile	6	0
179	WriteToFile	0	0
Total	179	179	179

Object CPartFile

	member attribute	protection	ODE4	ODE5
1	availablePartsCount	private	1	1
2	completedsize	private	2	3
3	confirmedsize	private	1	1
4	count	private	1	1
5	datarate	private	3	1
6	date	public	3	4
7	directory	protected	5	3
8	filehash	protected	0	15
9	filename	protected	8	14
10	filesize	protected	4	21
11	filetype	protected	0	0
12	fullname	private	1	4
13	lastpurgetime	private	2	3
14	lastsearchtime	private	3	2
15	m_ClientSrcAnswered	private	0	0
16	m_iGainDueToCompression	private	2	1
17	m_iLastTimeSourcesLoaded	private	1	2
18	m_iLastTimeSourcesLoadedPartial	private	2	2
19	m_iLastTimeSourcesSaved	private	2	2
20	m_iLoadingSourcesPartialTimeInterval	private	1	5
21	m_iLoadingSourcesTimeInterval	private	2	4
22	m_iLoadSourcesAtOnceLimit	private	1	4
23	m_iLostDueToCorruption	private	1	0
24	m_iMoviePreviewMode	public	2	2
25	m_iPermissions	public	2	4
26	m_iPriority	public	2	5
27	m_iRate	protected	2	2
28	m_iSavingSourcesTimeInterval	private	1	4
29	m_iSourceIDIndex	private	1	1
30	m_iSourceIndex	private	1	1
31	m_iTotalPacketsSavedDueToICH	private	1	1
32	m_LastNoNeededCheck	private	1	1
33	m_nCompleteSourcesAccuracy	inaccessible	2	0
34	m_nCompleteSourcesCount	inaccessible	2	0
35	m_nCompleteSourcesTime	inaccessible	2	2
36	m_nLastBufferFlushTime	private	1	1
37	m_nLastCompleteSrcCount	private	1	1
38	m_nSavedReduceDownload	private	1	1
39	m_nTotalBufferData	private	2	3
40	partmetfilename	private	1	3
41	percentcompleted	private	2	1
42	percentconfirmed	private	1	1
43	priority	private	2	3
44	status	private	4	7
45	tempdir	private	1	4
46	transfered	private	2	3
47	transferingsrc	private	3	1
Total	47	47	47	47

Bibliography

Abreu, F.B. and Carapuca, R. 1994, 'Object-Oriented Software Engineering: Measuring and Controlling the Development Process', Originally published in *Proceedings of the 4th International Conference on Software Quality*, McLean, VA, USA, October 1994, Revised version available at http://citeseer.nj.nec.com/brito94objectoriented.html, [Accessed 28 April 2002].

Abreu, F., Goulao, M. and Esteves, R. 1995, 'Toward the Design Quality Evaluation of Object-Oriented Software Systems', *Proceedings of the Fifth International Conference on Software Quality*, American Society for Quality Control, Austin, Texas, pp. 44-57.

Abreu, F.B. and Melo, W. 1996, 'Evaluating the Impact of Object-Oriented Design on Software Quality', *Proceedings of the 3rd International Software Metrics Symposium, METRICS '96*, IEEE, Berlin, Germany, pp. 90-99.

Abreu, F. and Goulao, M. 2001, 'Coupling and Cohesion as Modularization Drivers: Are we being over-persuaded?', *Fifth European Conference on Software Maintenance and Reengineering*, Lisbon, Portugal.

Ammann, M.M. and Cameron, R.D. 1994, 'Measuring Program Structure with Inter-Module Metrics', *Proceedings of the 18th Annual International Computer Software and Applications Conference (COMPSAC 94)*, IEEE Computer Society, pp. 139-144.

Anonymous Assessor, 2005, *Confidential Examiner's Report On A Thesis Submitted For A Higher Degree By Research : Student Christine McClean*, UTS University Graduate School.

Arisholm, E., Briand, L.C. and Foyen, A. 2004, 'Dynamic Coupling Measurement for Object-Oriented Software' *IEEE Transactions on Software Engineering*, vol. 30, no. 8, pp. 491-506.

Basili, V.R. 1988, 'The TAME Project: Towards Improvement-Oriented Software Environments', *IEEE Transactions on Software Engineering*, vol. 14, no. 6, pp. 758-773.

Bell, D. And Morrey, I. And Pugh, J. 1992, *Software Engineering: a programming approach*, 2nd edition, Prentice Hall International, UK.

Bieman, J.M. and Kang, B-K. 1995, 'Cohesion and Reuse in an Object-Oriented System', *Proceedings ACM Symposium on Software Reusability, SSR'95*, ACM Press, Seattle, Washington, USA, pp. 259-262.

Blaxter, L., Hughes, C. and Tight, M. 1996, *How to Research*, Open University Press, UK.

Briand, L.C., Morasca, S. and Basili, V.R. 1996, 'Property-Based Software Engineering Measurement', *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68-85.

Briand, L.C., Daly, J.W. and Wust, J.K. 1997b, A Unified Framework for Cohesion Measurement in Object-Oriented Systems [Online]. Available: http://www.iese.fhg.de/ISERN/pub/isern_biblio_tech.html [Accessed 28 November 1999].

Briand, L.C. and Daly, J.W. and Wust, J.K. 1999, 'A Unified Framework for Coupling Measurement in Object-Oriented Systems', *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91-121.

Briand, L.C., Morasca, S. and Basili, V.R., 1999, 'Defining and Validating Measures for Object-Based High-Level Design', *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 722-743.

Briand, L.C., Morasca, S. and Basili, V.R. 2002, 'An Operational Process for Goal-Driven Definition of Measures', *IEEE Transactions on Software Engineering,* vol. 28, no. 12, pp. 1106-1125.

Bunge, M. 1977, *Treatise on Basic Philosophy: Ontology I: The Furniture of the World,* Riedel, Boston, 1977, cited by (Chidamber and Kemerer 1994).

Bunge, M. 1979, *Treatise on Basic Philosophy: Ontology II: The World of Systems,* Riedel, Boston, 1979, cited by (Chidamber and Kemerer 1994).

Carmines, E.G. and Zeller, R.A. 1979, *Reliability and Validity Assessment*, Sage Publications Inc., California.

Chidamber, S.R. and Kemerer, C.F. 1994, 'A Metrics Suite for Object Oriented Design', *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476-493.

Chidamber, S.R., Darcy, D.P. and Kemerer, C.F. 1998, 'Managerial Use of Metrics for Object-Oriented Software: An Exploratory Analysis' *IEEE Transactions on Software Engineering*, vol. 24, no. 8, pp. 629-639.

Churcher, N.I. and Shepperd, M.J. 1995a, 'Comments on "A Metrics Suite for Object Oriented Design"', *IEEE Transactions on Software Engineering*, vol. 21, no. 3., pp. 263-265.

Churcher, N.I. and Shepperd, M.J. 1995b, 'Towards a Conceptual Framework for Object Oriented Software Metrics', *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 2, pp. 69-76.

Davis, P.J. and Hersh, R. 1986, *Descartes Dream: The World According to Mathematics*, Penguin Books Ltd., London.

Diamantopoulos, A. and Schlegelmilch, B.B. 1997, *Taking the Fear Out of Data Analysis*, The Dryden Press, London.

Ellis, B. 1966, *Basic Concepts of Measurement*, Cambridge University Press, Cambridge.

Eriksson, H-E. and Penker, M. 1998, *UML Toolkit*, John Wiley & Sons, USA.

Fenton, N. and Melton, A. 1990, 'Deriving Structurally Based Software Measures', *Journal of Systems and Software*, vol. 12, pp. 177-187.

Fenton, N. 1994, 'Software Measurement: A Necessary Scientific Basis', *IEEE Transactions on Software Engineering*, vol. 20, no. 3, pp. 199-206.

Fenton, N.E. 1995, *Software metrics: A Rigorous Approach*, International Thomson Computer Press, London.

Ferrett, L.K. and Offutt, J. 2002, 'An Empirical Comparison of Modularity of Procedural and Object-oriented Software', *Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems (ICECCS'02)*, IEEE Computer Society, pp 173-182.

Gasking, D. 1960, 'Clusters', *The Australian Journal of Philosophy*, vol. 38, no. 1, pp. 1-36.

Grassmann, W.K. and Tremblay, J-P. 1996, *Logic and Discrete Mathematics: A Computer Science Perspective*, Prentice-Hall, New Jersey.

Hawryszkiewycz, I.T. 1990, *Relational Database Design: An Introduction*, Prentice Hall, Australia.

Henderson-Sellers, B. 1996, *Object-Oriented Metrics: Measures of Complexity*, Prentice Hall Inc., New Jersey.

Hitz, M. and Montazeri, B. 1995, 'Measuring Coupling and Cohesion in Object Oriented Systems', *Proceedings International Symposium on Applied Corporate Computing*, Monterrey, Mexico.

Hitz, M. and Montazeri, B. 1996, 'Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective', *IEEE Transactions on Software Engineering*, vol 22, no 4, pp. 276-270.

Hoffman, P. 2001, *Perl for Dummies*, 3rd Edition, IDG Books Worldwide, Foster City, CA.

IEEE Computer Society 1993, "IEEE Standard for a Software Quality Metrics Methodology", *IEEE Std 1061-1992*, The Institute of Electrical and Electronic Engineers, New York.

IEEE Computer Society 1998, "IEEE Standard for a Software Quality Metrics Methodology", *IEEE Std 1061-1998*, Revision of IEEE Std 1061-1992, The Institute of Electrical and Electronic Engineers, New York.

Jeffery, R., Scott, L. 2002, 'Has Twenty-five Years of Empirical Software Engineering Made a Difference?', *Proceedings of the Ninth Asia-Pacific Software Engineering Conference (APSEC'02)*, pp. 539-546.

Kitchenham, B.A. 1996, *Software Metrics: Measurement for Software Process Improvement*, Blackwell Publishers Inc., Massachusetts.

Kitchenham, B.A., Hughes, R.T. and Linkman, S.G. 2001, 'Modeling Software Measurement Data', *IEEE Transactions on Software Engineering*, vol. 27, no. 9, pp. 788-804.

Kitchenham, B., Pfleeger, S.L. and Fenton, N. 1995, 'Towards a Framework for Software Measurement Validation', *IEEE Transactions on Software Engineering*, vol. 21, no. 12, pp. 929-943.

Kitchenham, B., Pfleeger, S.L. and Fenton, N. 1997, 'Reply to: Comments on "Towards a Framework for Software Measurement Validation", *IEEE Transactions on Software Engineering*, vol. 23, no. 3, p. 189.

Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., El Emam, K. and Rosenberg, J. 2002, 'Preliminary Guidelines for Empirical Research in Software Engineering', *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721-734.

Kitchenham, B.A. and Stell, J.G. 1997, 'The danger of using axioms in software metrics', *IEE Proceedings of Software Engineering*, vol. 144, no. 5-6, pp. 279-285.

Leaney, J., Rowe, D. & O'Neill, T. 2002 'Issues in the construction of new measures within the discipline of Open Systems', *Proceedings of the Ninth Asia-pacific Software Engineering Conference (APSEC '02)*, pp. 527-536.

Li, W. 1998, 'Another metric suite for object-oriented programming', *The Journal of Systems and Software 44*, pp. 155-162.

Lippman, S.B. 1993, *C++ Primer*, 2nd edition, Addison-Wesley Publishing Company, Massachusetts.

Meyer, B. 1988, *Object-Oriented Software Construction*, Prentice Hall, New Jersey.

Meyer, B. 1997, *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, New Jersey.

Microsoft 1997, Microsoft Access 97, Copyright © 1998-1996, Microsoft Corporation.

Morasca, S., Briand, L.C., Weyuker, E.J. and Zelkowitz, M.V. 1997, 'Comments on "Towards a Framework for Software Measurement Validation"', *IEEE Transaction on Software Engineering*, vol. 23, no. 3, pp. 187-188.

Myers, G.J. 1975, *Reliable Software Through Composite Design*, Mason/Charter Publishers Inc., USA.

Offen, R.J. and Jeffery, R. 1997, 'Establishing Software Measurement Programs', *IEEE Software*, vol. 14, no. 2, pp. 45-53.

Page-Jones, M. 1988, *The Practical Guide to Structured Systems Design*, 2nd edition, Prentice-Hall International, Englewood Cliffs, NJ, USA.

Pfleeger, S.L., Jeffery, R., Curtis, B. and Kitchenham, B. 1997, 'Status Report on Software Measurement', *IEEE Software*, vol. 14, no. 2, pp. 33-43.

Pressman, R.S. 1992, *Software Engineering: a practitioner's approach*, McGraw-Hill, Singapore.

Ross, K.A. and Wright, C.R.B. 1992, *Discrete Mathematics,* 3rd edition, Prentice-Hall Inc., New Jersey.

Scientific Toolworks Inc. 2003, *Understand for C++*, Available: http://www.scitools.com [Accessed 1 December 2003].

Shepperd, M. and Ince, D. 1993, *Derivation and Validation of Software Metrics*, Clarendon Press, Oxford.

Shlaer, S. And Mellor, S.J. 1992, *Object Lifecycles: Modeling the World in States*, Prentice-Hall Inc., Englewood Cliffs, New Jersey.

Sproull, N.L. 1995, *Handbook of Research Methods: A Guide for Practitioners and Students in the Social Sciences*, 2nd Edition, The Scarecrow Press, Inc., Metuchen, N.J.

Stevens, S.S. 1946, 'On the Theory of Scales of Measurement', *Science*, vol. 103, no. 2684, pp. 677-680.

Stevens, W.P., Myers, G.J. and Constantine, L.L. 1974, 'Structured Design', *IBM Systems Journal*, no. 2, pp.115-139.

Swift, L. 2001, *Quantitative Methods for Business, Management and Finance*, Palgrave Publishers Ltd. New York.

Szyperski, C. 1998, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, Essex, England.

Tang, M-H and Chen, M-H 2002, 'Measuring OO Design Metrics from UML', *Lecture Notes in Computer Science*, vol. 2460/2002, Springer-Verlag Heidelberg, pp368-382.

The Australian Pocket Oxford Dictionary, 5th edition, , 2002, editied by Bruce Moore, Oxford University Press, Victoria, Australia.

Weyuker, E.J. 1988, 'Evaluating Software Complexity Measures', *IEEE Transactions on Software Engineering*, vol. 14, no 9, pp. 1357-1365.

Wolberg, G. 1990, *Digital Image Warping*, IEEE Computer Society Press, Los Alimitos, California.

Wolfe, J. 2003, *How to Write a PhD Thesis*, [Online]. Available: http://www.phys.unsw.edu.au/~jw/thesis.html [Accessed 18 September 2004].

Yourdon, E., and Constantine, L. 1979, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, Englewood Cliffs, N.J.

Zuse, H. 1996, 'Foundations of Object-oriented Software Measures', *IEEE Proceedings of METRICS '96*, pp. 75-87.