

# **Integrating Measurement Techniques in an Object-Oriented Systems Design Process**

**Philippe Li-Thiao-Té**

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

December 1999

**Supervisors: Prof. Jessie Kennedy and Dr. John Owens**

Date of viva: April 20<sup>th</sup>, 2000

Internal examiner: **Dr. Pete Barclay**

External examiner: **Dr. Dilip Patel**

**Napier University, School of Computing, Edinburgh, Scotland**

## **Acknowledgements**

I wish to express my thanks and acknowledge the assistance to people without whom this thesis would not exist.

Firstly thanks are due to my supervisors Prof. Jessie Kennedy and Dr. John Owens, both lecturers of Napier University, Edinburgh, Scotland. I am most grateful to Jessie for her patience, for her valuable insights in Object-Oriented modelling and database areas. Many of our intense discussions provided me with focus in my research work and her thorough proofreading and corrections have contributed to the accuracy of content and style of the thesis. Overall, her support throughout the year has made this work enjoyable and possible. I would like to thank John for his valuable advice and for sharing his in-depth knowledge in various areas of object technology. His guidance on the aims and directions of the thesis permitted me to tackle the obscure and controversial object-oriented mechanism of method redefinition.

I offer especial thanks to Prof. Mike Jackson of Wolverhampton University for numerous discussions, useful feedback and criticisms on various aspects of Object Oriented design as well as the measurement technique areas. His views on many parts of the thesis have helped my understanding of the obscure issues since the early days of this work. I am also grateful to my internal and external examiners, respectively Dr. Pete Barclay of Napier University and Dr. Dilip Patel of South Bank University, for their valuable comments which contributed to the improvement of the overall quality of this thesis.

I am indebted in many respects to all the staff, academic, research, support and administrative in the School of Computing at Napier University which provided me with financial support and time for writing-up.

I am grateful to all of Arclight Strategy Systems Ltd. staff who gave me support during the last phases of the writing-up.

I would like to thank my friends for their support and for the many distracting discussions that kept me alert and aware of numerous aspects of life. Particular thanks are due to Alan Garny for his endless cynicism, Cédric Raguenaud for his strong views on a non Object-Oriented world, my cousin Jean-Luc Haw-Kwan-Yuen for his perspectives on a simple life in a complex society!

I would like to offer warm thanks to my fiancée Agnès Laï, so far but so close, who accompanies me and fills my life with happiness.

Finally and most of all I would like to thank my family for their forbearance and their support during these long years away from home. I am most grateful to my parents and my sister Brigitte and her family for giving me inspirations and strength throughout my life.

I have special thoughts for late members of my family whom I missed from the outset of my studies. Especially, I have found memories for my grandad Koung-Koung who taught me honour, loyalty, respect and proudness in life.

Thank you to you all.

*Edinburgh, 19<sup>th</sup> December 1999*

The next section gives the same acknowledgements in French.

## Remerciements

Je voudrais remercier et exprimer ma reconnaissance aux personnes sans qui cette thèse n'existerait pas.

Tout d'abord, je dois remercier mes directeurs de thèse Prof. Jessie Jennedy et Dr. John Owens, respectivement maître de conférences et professeur à "Napier University", Edimbourg, Ecosse. Je suis particulièrement reconnaissant à Jessie pour sa patience et ses idées pertinentes aussi bien en conception orientée-objets que dans le domaine des bases de données. D'innombrables et intenses discussions m'ont permis de me concentrer sur mon travail de recherche et ses méticuleuses relectures accompagnées de corrections ont contribué à la rigueur du contenu et du style de cette thèse. Plus généralement, ses encouragements durant les années ont rendu ce travail agréable et possible. Je voudrais remercier John pour ses recommandations et pour avoir partagé avec moi sa connaissance approfondie dans divers domaines des technologies orientées-objets. Ses conseils concernant les objectifs et la direction de la thèse m'ont permis d'aborder l'ambigu et polémique principe de redéfinition des méthodes dans les systèmes orientés-objets.

J'offre des remerciements spéciaux au Prof. Mike Jackson de "Wolverhampton University" pour les discussions, les commentaires et les critiques sur de nombreux aspects de la conception orientée-objets mais aussi du domaine des techniques de mesures. Ses opinions sur plusieurs parties de cette thèse m'ont aidé à comprendre les problèmes les plus difficiles depuis le début de ce travail. Je suis aussi reconnaissant à mes examinateurs interne et externe, respectivement Dr. Pete Barclay de "Napier University" and Dr. Dilip Patel de "South Bank University", pour leurs précieux commentaires qui ont contribué à l'amélioration de la qualité générale de la thèse.

Je suis reconnaissant au personnel, académique, chercheur, support technique et administratif du département "School of Computing" de "Napier University" qui m'ont financé et accordé du temps pour l'écriture de la thèse. Merci à tout le personnel d'"Arclight Strategy Systems Ltd." pour m'avoir soutenu pendant les dernières phases d'écriture de la thèse.

Je voudrais remercier mes amis pour leur support et pour les nombreuses discussions divertissantes qui m'ont permis de rester attentif et au fait des divers aspects de la vie. En particulier, merci à Alan Garny pour son interminable cynisme, à Cédric Raguenaud pour ses vues déterminées sur un monde non-orienté-objets, à mon cousin Jean-Luc Haw-Kwan-Yuen pour ses perspectives d'une vie simple dans une société compliquée!

Je voudrais offrir des remerciements les plus chaleureux à ma fiancée Agnès Laï, si loin mais si proche, pour m'accompagner et remplir ma vie de bonheur.

Finalement et plus que tout, je tiens à remercier toute ma famille pour leur patience and leur soutien pendant ces longues années loin de la maison. Je suis reconnaissant à mes parents, ma soeur Brigitte et sa famille pour me donner l'inspiration et la force tout au long de ma vie.

Des pensées spéciales vont aux regrettés membres de ma famille que j'ai beaucoup manqué lors de mes études. Particulièrement, j'ai de chers souvenirs pour mon grand-père Koung-Koung qui m'a enseigné honneur, loyauté, respect et fierté dans la vie.

Merci à vous tous.

*Edimbourg, le 19 Décembre 1999*

*To my parents*

## Abstract

The theme of this thesis is the assessment of quality in class hierarchies. In particular, the notion of inheritance and the mechanism of redefinition from a modelling perspective are reviewed. It is shown that, in Object-Oriented languages, controversial uses of inheritance can be implemented and are subject of debate as they contradict the essence of inheritance. The discovery of an unexpected use of the method redefinition mechanism confirmed that potential design inconsistencies occur more often than expected in class hierarchies. To address such problems, design heuristics and measurement techniques are investigated as the main instrument tools for the evaluation "goodness" or "badness" in class hierarchies. Their benefits are demonstrated within the design process.

After the identification of an obscure use of the method redefinition mechanism referred to as the *multiple descendant redefinition* (MDR) problem, a set of metrics based on the GQM/MEDEA [Bri&al94] model is proposed. To enable a measurement programme to take place within a design process, the necessary design considerations are detailed and the technical issues involved in the measurement process are presented. Both aspects form a methodological approach for class hierarchy assessment and especially concentrate on the use of the redefinition mechanism.

As one of the main criticisms of the measurement science is the lack of good design feedback, the analysis and interpretation phase of the metrics results is seen as a crucial phase for inferring meaningful conclusions. A novel data interpretation framework is proposed and includes the use of various graphical data representations and detection techniques. Also, the notion of redefinition profiles suggested a more generic approach whereby a pattern profile can be found for a metric. The benefits of the data interpretation method for the extraction of meaningful design feedback from the metrics results are discussed.

The implementation of a metric tool collector enabled a set of experiments to be carried out on the Smalltalk class hierarchy. Surprisingly, the analysis of metrics results showed that method redefinition is heavily used compared to method extension. This suggested the existence of potential design inconsistencies in the class hierarchy and permitted the discovery of the MDR problem on many occasions. In addition, a set of experiments demonstrates the benefits of example graphical representations together with detection techniques such as *alarms*. In the light of facilitating the interpretation phase, the need for additional supporting tools is highlighted.

This thesis illustrates the potential benefits of integration of measurement techniques within an Object-Oriented design process. Given the identification of the MDR problem, it is believed that the redefinition metrics are strong and simple candidates for detecting complex design problems occurring within a class hierarchy. An integrated design assessment model is proposed which logically fits into an incremental design development process. Benefits and disadvantages of the approach are discussed together with future work.

## Table of contents

<b>GLOSSARY OF TERMS, ABBREVIATIONS, NOTATIONS AND TRADEMARKS .....</b>	<b>15</b>
<b>1. INTRODUCTION .....</b>	<b>19</b>
<b>2. BACKGROUND.....</b>	<b>25</b>
2.1. INHERITANCE AND ASSOCIATED PROBLEMS.....	27
2.1.1. <i>Use of inheritance.....</i>	27
2.1.2. <i>Class hierarchy organisation .....</i>	28
2.1.3. <i>Subclassing, subtyping or specialising.....</i>	29
2.1.4. <i>Usability and extensibility .....</i>	31
2.1.5. <i>Property inheritance scheme definition.....</i>	34
2.1.6. <i>Property ownership transfer.....</i>	35
2.1.7. <i>Encapsulation: visibility and accessibility of properties .....</i>	36
2.1.8. <i>Consequences of encapsulation on the inheritance scoping control .....</i>	38
2.1.9. <i>Common inheritance design mistakes .....</i>	41
2.2. ON THE NOTION OF REDEFINITION .....	47
2.2.1. <i>The redefinition principle.....</i>	48
2.2.2. <i>Conditions for realising method redefinition.....</i>	49
2.2.3. <i>Descendants' heritage extent (hierarchy collapse) .....</i>	50
2.2.4. <i>The main redefinition variants .....</i>	52
2.2.5. <i>Remark on super method calls.....</i>	53
2.2.6. <i>Disinheritance and inheritance refusal .....</i>	54
2.3. HEURISTICS OR GUIDELINES FOR OBJECT-ORIENTED DESIGN.....	56
2.3.1. <i>Definition and purpose.....</i>	57
2.3.2. <i>Interpretation.....</i>	58
2.3.3. <i>Example of heuristic's application.....</i>	60
2.4. ASSESSMENT TECHNIQUES .....	63
2.4.1. <i>Roles of technical measurement .....</i>	64
2.4.2. <i>Software quality model.....</i>	65
2.4.3. <i>Properties of software measures .....</i>	67
2.4.4. <i>Internal quality factors of OO design.....</i>	68
2.4.5. <i>Data availability and metrics collection.....</i>	70
2.4.6. <i>Metrics interpretation.....</i>	71
2.4.6.1. <i>Remark on the dependencies between metrics .....</i>	74
<b>3. ASSESSING THE PROPERTIES INHERITANCE SCHEME FOR THE MULTIPLE DESCENDANT REDEFINITION PROBLEM IN OBJECT-ORIENTED SYSTEMS .....</b>	<b>76</b>
3.1. METHOD REDEFINITION: USES AND ABUSES .....	79
3.1.1. <i>Method redefinition in class hierarchies .....</i>	79
3.1.2. <i>Multiple descendant redefinition (MDR) problem.....</i>	80
3.1.3. <i>Example inheritance hierarchy that avoids the MDR problem .....</i>	81
3.1.4. <i>Descendants heritage extent for the MDR problem.....</i>	84
3.2. MEASURING REDEFINITION IN OBJECT-ORIENTED SYSTEMS.....	85
3.2.1. <i>The method redefinition assessment .....</i>	85
3.2.2. <i>Percentage of redefined methods per level within a hierarchy (PRMH).....</i>	87
3.3. DESIGN CONSIDERATIONS FOR INHERITANCE ASSESSMENT .....	88
3.3.1. <i>Methodological approach for class hierarchy assessment.....</i>	89
3.3.2. <i>A design information repository with metaclass facilities .....</i>	90
3.3.3. <i>Class analysers.....</i>	94
3.3.4. <i>State transition diagram for the method redefinition mechanism.....</i>	99
3.3.4.1. <i>Remark: method redefinition and unexpected message sends.....</i>	102
3.3.5. <i>Behavioural inheritance analysis .....</i>	105
3.3.5.1. <i>Experiments on the Collection class.....</i>	106
3.4. MECHANISMS FOR DATA INTERPRETATION OF METRICS FOR OBJECT-ORIENTED SYSTEMS.....	108
3.4.1. <i>Introduction.....</i>	108
3.4.2. <i>Motivation and approach for interpretation.....</i>	109
3.4.3. <i>Metrics interpretation framework.....</i>	111
3.4.3.1. <i>Designers' perceptions and decisions.....</i>	112
3.4.3.2. <i>Raw data representation .....</i>	113
3.4.3.3. <i>Profile analysis and design feedback.....</i>	116
3.4.3.4. <i>Factors affecting the interpretation process.....</i>	118



3.5.	CONCLUSION.....	119
<b>4.</b>	<b>METRIC TOOL COLLECTOR AND IMPLEMENTATION ISSUES .....</b>	<b>120</b>
4.1.	INTRODUCTION.....	120
4.2.	REQUIREMENTS.....	121
4.2.1.	<i>Features</i> .....	121
4.3.	ANALYSIS AND DESIGN OF THE METRIC COLLECTOR TOOL.....	122
4.3.1.	<i>Class lineage and parsing strategies</i> .....	122
4.4.	ARCHITECTURE.....	124
4.5.	USER INTERFACES.....	126
4.5.1.	<i>The System Metric Browser</i> .....	126
4.5.2.	<i>Metrics derivation</i> .....	127
4.5.3.	<i>The method profiles manager</i> .....	128
4.5.3.1.	<i>The method profiles browser</i> .....	130
4.5.4.	<i>The definition of ranges for the alarmer</i> .....	133
4.6.	CONCLUDING REMARKS.....	135
<b>5.</b>	<b>EXPERIMENTS.....</b>	<b>137</b>
5.1.	OVERVIEW OF THE METHOD REDEFINITION PROFILES USING THE PRM METRIC.....	138
5.2.	SMALLTALK OBJECT HIERARCHY.....	141
5.3.	COLLECTION BRANCH AND STREAM BRANCH.....	142
5.4.	WINDOWBUILDER PRO/V BRANCH.....	144
5.4.1.	<i>GraphicObject branch</i> .....	145
5.5.	T-GEN SYSTEM.....	146
5.5.1.	<i>T-gen system redefinition profile</i> .....	148
5.5.2.	<i>T-gen: TreNode branch redefinition profile</i> .....	149
5.5.3.	<i>T-gen: AbstractScanner branch redefinition profile</i> .....	151
5.6.	CUMULATIVE MEASURE FOR THE COLLECTION, STREAM, OBJECT AND GRAPHICOBJECT BRANCHES.....	153
5.7.	EFFECTS OF THE T-GEN SYSTEM ON THE SMALLTALK HIERARCHY.....	156
5.8.	EFFECTS OF THE T-GEN SYSTEM ON THE COLLECTION BRANCH REDEFINITION PROFILE.....	157
5.9.	METRIC RESULTS VISUALISATION AND INTERPRETATION.....	159
5.9.1.	<i>Surface bar charts</i> .....	160
5.9.2.	<i>Surface charts</i> .....	161
5.9.3.	<i>Addition bar charts</i> .....	162
5.9.4.	<i>Radar charts</i> .....	162
5.9.5.	<i>A colour coded range bar charts</i> .....	163
5.9.6.	<i>Visualisation uses</i> .....	164
5.10.	THE CONCEPT OF "ALARMERS".....	165
5.11.	DATA INTERPRETATION SYSTEM.....	168
5.12.	CONCLUSION OF THE EXPERIMENTS.....	168
<b>6.</b>	<b>DISCUSSION AND CONCLUSION.....</b>	<b>172</b>
	<b>REFERENCES.....</b>	<b>187</b>
<b>A.</b>	<b>APPENDIX.....</b>	<b>200</b>
A.1.	HEURISTICS' CLASSIFICATION.....	200
A.2.	DETAILED DESIGN OF THE MAIN COMPONENTS OF THE METRIC PROTOTYPE TOOL.....	201
A.2.1.	<i>Basic metrics repository</i> .....	201
A.2.2.	<i>Dictionary structures for metrics</i> .....	202
A.2.3.	<i>A persistent repository structure</i> .....	204
A.2.4.	<i>The profile manager</i> .....	206
A.2.5.	<i>The metric engine</i> .....	208
A.2.6.	<i>The hierarchy browser and profile manager designs</i> .....	208
A.2.7.	<i>The method profiles browser</i> .....	213
A.3.	REMARKS ON THE CONSEQUENCES OF THE ENCAPSULATION MECHANISM.....	214

## List of figures

---

<i>Figure 1.1: Objectives of the research work</i> .....	22
<i>Figure 1.2: Measure of level of redefinition in the Smalltalk Object hierarchy</i> .....	23
<i>Figure 2.1: Subclassing (1), subtyping (2) and specialisation (3) hierarchies</i> .....	30
<i>Figure 2.2: Class properties</i> .....	35
<i>Figure 2.3: Transfer of property ownership in an inheritance hierarchy</i> .....	36
<i>Figure 2.4: Example of transfer of property ownership</i> .....	36
<i>Figure 2.5: Property modifiers in OO programming languages</i> .....	37
<i>Figure 2.6: Stream hierarchy with multiple inheritance</i> .....	40
<i>Figure 2.7: Traversal paths for single and multiple inheritance</i> .....	41
<i>Figure 2.8: Coupling with instance variable</i> .....	44
<i>Figure 2.9: Coupling with method</i> .....	44
<i>Figure 2.10: Coupling with method signature</i> .....	44
<i>Figure 2.11: Coupling with inheritance</i> .....	45
<i>Figure 2.12: ENGINE class</i> .....	45
<i>Figure 2.13: ENGINE hierarchy</i> .....	45
<i>Figure 2.14: Expected descendant heritage extent</i> .....	51
<i>Figure 2.15: Part of the Smalltalk Collection branch</i> .....	52
<i>Figure 2.16: Different types of methods redefinition</i> .....	53
<i>Figure 2.17: Three possible designs for the class Person</i> .....	60
<i>Figure 2.18: Mapping and modelling gap</i> .....	61
<i>Figure 2.19: A company information system</i> .....	62
<i>Figure 2.20: The GQM/MEDEA model</i> .....	66
<i>Figure 3.1: Object-oriented design assessment model</i> .....	77
<i>Figure 3.2: Smalltalk hierarchy redefinition profile</i> .....	80
<i>Figure 3.3: Life history of the includes: redefined method in the Smalltalk Collection branch</i> .....	81
<i>Figure 3.4: MDR and code duplication in the Stream class hierarchy</i> .....	82
<i>Figure 3.5: Stream hierarchy using mixins classes</i> .....	83
<i>Figure 3.6: Descendant heritage extent with MDR anomaly</i> .....	84
<i>Figure 3.7: Complexity metrics at hierarchy level</i> .....	87
<i>Figure 3.8: Meta-model of main OO concepts</i> .....	91
<i>Figure 3.9: Tree parsing strategy</i> .....	95
<i>Figure 3.10: Name space collisions with multiple inheritance</i> .....	96
<i>Figure 3.11: Class wrapper</i> .....	97
<i>Figure 3.12: Hierarchy wrapper</i> .....	98
<i>Figure 3.13: State-chart diagram for method redefinition</i> .....	100
<i>Figure 3.14: Expected method invocation</i> .....	102
<i>Figure 3.15: Examples of unexpected method invocations</i> .....	103
<i>Figure 3.16: Distant MDR scenarios</i> .....	104
<i>Figure 3.17: Method life history representation</i> .....	105

<i>Figure 3.18: Method life history for the Collection branch</i> .....	106
<i>Figure 3.20: Analysis, interpretation and interactions</i> .....	113
<i>Figure 3.21: Data representation</i> .....	114
<i>Figure 3.22: Profile analysis</i> .....	116
<i>Figure 3.23: Interpretation factors</i> .....	118
<i>Figure 4.1: Levels of derivation</i> .....	123
<i>Figure 4.2: Parsing strategies in class hierarchies</i> .....	124
<i>Figure 4.3: Metric collector tool architecture</i> .....	125
<i>Figure 4.4: Roadmap for user interfaces presentation</i> .....	126
<i>Figure 4.5: Prototype metric tool main window</i> .....	127
<i>Figure 4.6: Redefinition metric at system level</i> .....	128
<i>Figure 4.7: Method profile list manager</i> .....	129
<i>Figure 4.8: Redefined methods browser</i> .....	130
<i>Figure 4.9: Features of the methods browser</i> .....	131
<i>Figure 4.10: Method senders</i> .....	132
<i>Figure 4.11: Method implementors</i> .....	133
<i>Figure 4.12: Alarmer ranges definition</i> .....	134
<i>Figure 4.13: System Metric Browser with alarmer display</i> .....	135
<i>Figure 5.1: PRM for the Smalltalk Object hierarchy</i> .....	139
<i>Figure 5.2 (a) and (b): PRM for the WindowBuilder Pro/V and T-gen systems</i> .....	139
<i>Figure 5.3 (a) and (b): PRM for the Collection and Stream branches</i> .....	139
<i>Figure 5.4: PRM for the GraphicObject branch</i> .....	139
<i>Figure 5.5 (a) and (b): PRM for the TreNode and AbstractScanner branches</i> .....	140
<i>Figure 5.6 (a) and (b): PRM for the Object and Collection hierarchies with the T-gen system installed</i> .140	
<i>Figure 5.7: PCRM and PEM for the Object hierarchy</i> .....	141
<i>Figure 5.8 (a) and (b): PCRM and PEM for the Collection and Stream hierarchies</i> .....	142
<i>Figure 5.9 (a) and (b): Collection branch at DIT = 3 and FileStream at DIT=4</i> .....	143
<i>Figure 5.10: Collection method profile</i> .....	144
<i>Figure 5.11: PCRM and PEM for the WindowBuilder Pro/V</i> .....	144
<i>Figure 5.12: PCRM and PEM for GraphicObject branch</i> .....	145
<i>Figure 5.13: GraphicObject method profile</i> .....	146
<i>Figure 5.14: T-gen: ItemSet class redefinition profile</i> .....	148
<i>Figure 5.15: PCRM and PEM for the T-gen system</i> .....	149
<i>Figure 5.16: T-gen: PCRM and PEM for the TreNode branch</i> .....	149
<i>Figure 5.17: TreNode method profile</i> .....	151
<i>Figure 5.18: T-gen: PCRM and PEM for the AbstractScanner branch</i> .....	152
<i>Figure 5.19: AbstractScanner method profile</i> .....	152
<i>Figure 5.20: Cumulative PRM for the Collection branch</i> .....	153
<i>Figure 5.21: Number of classes per DIT level</i> .....	153
<i>Figure 5.22: Number of methods per DIT level</i> .....	154
<i>Figure 5.23: Cumulative PRM for the Object branch</i> .....	154
<i>Figure 5.24: Cumulative PRM for the GraphicObject branch</i> .....	154
<i>Figure 5.25: Subset of GraphicObject subclasses branch at DIT=3</i> .....	155

<i>Figure 5.26: Smalltalk Object hierarchy with the T-gen system installed.....</i>	<i>156</i>
<i>Figure 5.27 (a) and (b): PCRM and PEM for the Collection hierarchy with the T-gen system installed..</i>	<i>158</i>
<i>Figure 5.28 (a) and (b): Surface bar profiles for the Object and GraphicObject branches.....</i>	<i>160</i>
<i>Figure 5.29 (a) and (b): Surface profiles for the Object and GraphicObject branches .....</i>	<i>161</i>
<i>Figure 5.30 (a) and (b): Addition bar charts profiles for the Object and GraphicObject branches .....</i>	<i>162</i>
<i>Figure 5.31 (a) and (b): Radar charts profiles for Object and GraphicObject branches.....</i>	<i>162</i>
<i>Figure 5.32 (a) and (b): Colour coded bar for the Object and GraphicObject branches .....</i>	<i>163</i>
<i>Figure 5.33: Data interpretation system.....</i>	<i>168</i>
<i>Figure 6.1: Modelling and assessment tasks.....</i>	<i>181</i>
<i>Figure 6.2: Incremental Design and Assessment Process.....</i>	<i>184</i>
<i>Figure 6.3: Integrated model for design and assessment.....</i>	<i>184</i>
<i>Figure A.1: Dictionary of redefined methods per class.....</i>	<i>202</i>
<i>Figure A.2: Dictionary for the total number of methods per class.....</i>	<i>203</i>
<i>Figure A.3: Dictionary for replaced and extended methods .....</i>	<i>204</i>
<i>Figure A.4: Persistent repository model .....</i>	<i>205</i>
<i>Figure A.5: Profile manager model .....</i>	<i>206</i>
<i>Figure A.6: The hierarchy browser and profile manager designs .....</i>	<i>208</i>
<i>Figure A.7: Metric engine model .....</i>	<i>209</i>
<i>Figure A.8: The method profiles browser design.....</i>	<i>213</i>

**List of tables**

---

<i>Table 2.1: Identification of objects from textual specifications</i> .....	61
<i>Table 2.2: GQM levels</i> .....	65
<i>Table 3.1: Class design features</i> .....	93
<i>Table 3.2: Attribute design features</i> .....	93
<i>Table 3.3: Method design features</i> .....	94
<i>Table 3.4: Inheritance paths table</i> .....	95
<i>Table 3.5: State transition table for method redefinition</i> .....	101
<i>Table 3.6: Smalltalk Express Object branch redefinition profile</i> .....	115
<i>Table 5.1: List of assessed hierarchies</i> .....	138
<i>Table 5.2: Example of equally distributed ranges</i> .....	163
<i>Table 5.3: Summary of visualisation types</i> .....	165
<i>Table A.1: Smalltalk metaclass information</i> .....	201
<i>Table A.2: Allowed property modifiers for a redefined method in Java</i> .....	214

## Conference papers

---

[Ltt&al97a]

P. Li-Thiao-Té, J. Kennedy and J. Owens, "Mechanisms for Data Interpretation of Metrics for OO Systems", TOOLS Asia '97 Conference proceedings, Beijing, China , Sept. 1997.

[Ltt&al97b]

P. Li-Thiao-Té, J. Kennedy and J. Owens, "Assessing Inheritance for the Multiple Descendant Redefinition Problem in OO Systems", OOIS '97 Conference proceedings, Brisbane, Australia, Nov. 1997.

## Glossary of terms, abbreviations, notations and trademarks

---

Object-oriented technology introduced many concepts which have been interpreted differently with the research community. This section describes the meaning given to the technical terms used in the thesis so as to avoid confusion. For convenience, the "Smalltalk notation" will be adopted in most cases unless differently stated.

### Terms

- Originally, *properties* or *attributes* were used to describe characteristics of entities e.g. Entity-relationship model [Chen76]. In the OO paradigm, it is commonly understood that they represent both the instance variables and behaviour of the class.
- A *behaviour* or *service* for a class corresponds to a:
  - *method* in Smalltalk and Java,
  - *member function* in C++,
  - *feature* in Eiffel.
- An *instance variable* in Smalltalk is the same as a local variable in C++ within a class definition.
- A *class variable* in Smalltalk is the equivalent of a static variable in C++ within a class definition.
- The declaration of the method name and arguments list is referred to as *the signature of the method*. For typed systems, it also encompasses the return type of the method.
- *Method redefinition* is also known as *method overriding*. *Name overloading* is different than redefinition in that it refers to the different signatures for the same method which are bound at run-time.
- A *pure virtual function* in C++ is known as *deferred function* in Eiffel.
- A *setter* or *getter* designates a method which, respectively, sets the value of an attribute or gets the value of the attribute.
- *Methodology*: An organised, documented set of procedures and guidelines for one or more phases of the software life cycle e.g. analysis or design. Many methodologies include a diagramming notation for documenting the results of the procedure; a step-by-step "cookbook" approach for carrying out the procedure; and an objective (ideally quantified) set of criteria for determining whether the results of the procedure are of acceptable quality.
- The term *process* is understood as a defined set of activities to undertake to realise an objective.

- The *Smalltalk image* refers to the Smalltalk class library and some applications. When Smalltalk is started, the Smalltalk executable system uploads the image in memory. An image mainly consists of two files: the `sources.sml` file that contains all the source code and the `change.log` that holds all recent user changes to the image. For any code changes, the system re-compiles the source code into byte code that can be executed by the Smalltalk virtual machine [GolRob85].

### Abbreviations

- API : Application Programming Interface
- CASE: Computer-Aided Software Engineering
- DIT: Depth of Inheritance Tree
- ER: Entity-Relationship
- IDE: Integrated Development Environment
- OMT: Object Modelling Technique method created by Rumbaugh [Rum91].
- OO: object-oriented
- OOD: object-oriented design
- OOM: object-oriented modelling

### Notations

- **Level in inheritance**

By convention, the depth of inheritance is numbered from the root class to its leaves starting from 0.

- **Class property description**

A description of a class, at depth of inheritance  $l$  can be defined as the description of its properties i.e.  $C_l = \{\text{variables}, \text{methods}\}$  where  $\text{variables} = |\text{inst}_1, \text{inst}_2, \dots, \text{inst}_n|$  and  $\text{methods} = \langle \text{mth}_1(), \text{mth}_2(), \dots, \text{mth}_n() \rangle$



Example:

The notation  $C_2=\{\text{instA}, \text{instB}, \langle \text{mthA}(), \text{mthB}() \rangle\}$  means that a class  $C$  is situated at level 2 in the hierarchy, holds 2 instance variables  $\text{instA}$ ,  $\text{instB}$ , and two instance methods  $\text{mthA}()$ ,  $\text{mthB}()$ . The parameter list will be given when necessary.

- **Class property description with inherited features**

There will be cases where some or all inherited properties have to be shown for a class. The purpose of inserting inherited features in the notation will be mainly used to describe methods which are redefined in a subclass. Thus, a class holding inherited methods i.e. methods defined at least once in one of its parents will be noted

$C = \{[\text{inheritedVariables}], \text{variables}, [\text{inheritedMethods}], \text{methods}\}$

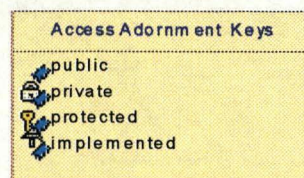
where

$\text{inheritedVariables} = \|\text{inst}_1, \text{inst}_2, \dots, \text{inst}_n\|$ ,  $\text{inheritedMethods} = \langle \langle \text{mth}_1(), \text{mth}_2(), \dots, \text{mth}_n() \rangle \rangle$

and  $\text{inheritedVariables} \cap \text{variables} = \emptyset$ ,  $\text{inheritedMethods} \cap \text{methods} = \emptyset$ . The "[...]" denotes the fact that the properties are optionally mentioned when using the notation. When a notation contains inherited methods i.e.  $\langle \langle \dots \rangle \rangle$ , the listed methods physically exist in the subclass which means that those methods are the ones redefined and therefore inherited. Redefined methods constitute part of the additional properties of a class.

- **Properties access adornment**

The Rational Rose 98<sup>1</sup> case tool defines access adornments to specify the type of access allowed between classes, as well as on attributes, operations and roles. There are four types of access adornments: public, private, protected, or implementation and are represented with the graphical symbol appearing in front of the properties as follows:



**Public:** Public access means that the members of a class are accessible to all clients.

<sup>1</sup> Rational Rose 98, Rational Enterprise Edition, Copyright © 1991-1998, Rational Software Corporation, All Rights Reserved, Portions ©, 1992-1998, Summit Software Company, <http://www.rational.com>

**Protected:** Protected access means that the members of a class are accessible only to subclasses, friends, or to the class itself.

**Private:** Private access means that the members of a class are accessible only to the class itself or to its friends.

**Implementation:** Implementation access means that the members of a class in a package **P** are accessible only by classes that import the package **P**.

- **Inheritance relationship**

If a class **B** is a subclass of **A**, the inheritance relationship is denoted  $B < A$ ; therefore the depth of inheritance of  $A <$  depth of inheritance of **B**.

### **Trademarks**

- Sun Microsystems, Java and Java Development Kit are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. <http://www.sun.com>.
- IBM® is a registered trademark of IBM in the United States. <http://www.ibm.com>.
- Microsoft, PowerBuilder, Microsoft Foundation Class (MFC), Windows, PowerPoint, Excel, the Wizard function and Object Linking and Embedding are trademarks or registered trademarks of Microsoft. <http://www.microsoft.com>.
- ENVY is a registered trademark of Object Technology International Inc. (OTI).
- Rational and Rational Rose are registered trademarks of Rational Software Corporation in the United States and in other countries.
- Eiffel and Design by Contract™ are trademarks of Interactive Software Engineering.
- Borland® C++ is a registered trademark of Inprise Corporation
- ITASCA is a registered trademark of IBEX Computing SA
- Franz and Allegro CLOS language are registered trademark of Franz®, Inc. <http://www.franz.com>

## 1. Introduction

*"Teaching kids to count is fine, but teaching them what counts is best" – Bob Talbert*

The design of software applications using object-oriented (OO) concepts and techniques is a challenging process where creativity, risk, uncertainty, experience, judgement and good sense predominate. Many factors determine the success of application development. Current OO design methods provide the designers with a logical and progressive set of tasks and techniques permitting the discovery of many candidate design solutions to a problem. However, there are still no reliable ways or *"no teachable step-by-step rules"* [Mey97] for producing good design. In the final decision making process, the designers' experiences and knowledge determine the choice of the design solution. This choice reflects the degree of satisfaction of the requirements and criteria of the problem, thus the notion of *design trade-off*. The design choices directly affect the future of a project.

Object technology provides designers with invaluable concepts and techniques that improve the software development process. Examples of benefits include a better capture and modelling of the business requirements. Similarly, the software applications produced gain benefits from their degree of reusability, maintainability and adaptability to new requirements. Overall, such benefits reduce the cost of the development. To date, the current push for object technology on the market is significant. In many areas of computing such as object-relational databases, knowledge management or Internet based applications, the adoption of OO design methods and OO programming languages have proven useful in building successful software applications. How reproducible those experiences are in a different context is unknown. Software engineers have also learned from unsuccessful experiences. Although object-oriented software development has existed for decades, several fundamental aspects of object computing are, however, still the subject of debate and are actively researched [AskBer92, Sho&a193, Web95]. Essentially, the issues relate to the appropriateness of the OO concepts to tackle complex requirements of business applications.

Architectural issues are one of the major aspects of software design. For software to be modular, one possible approach is to decompose complex problems into simpler sub-problems. In such a way, the identification of modules is made easier and the important OO aspect of separation of concerns is realised. The fundamental unit of construction in OO design methods is the notion of *object*. It combines data and behaviour into a coherent entity. Each object represents a unique concept in the real world. When objects are assembled together, coherent abstractions of real-world problems are formed and the co-operation between objects permits the realisation of the features of the application. The abstractions of an OO model are discovered during the generalisation process. Overall, abstraction in an object model contributes to the desired extensibility and reusability aspects of the components.

*Inheritance* is the OO concept that permits the abstraction of objects. From a conceptual point of view, inheritance is the mechanism by which a class referred to as a *subclass* conforms to another class, its *superclass*, thereby forming a class hierarchy. Conceptually, the subclass can be seen as a specialisation of its parent class. Pragmatically, from a software engineering perspective, authors have also expressed inheritance as a mechanism for code sharing and code reuse. In a class hierarchy, the parent classes provide properties that are inherited by their subclasses. Although it is generally recognised that inheritance is one of the major aspects of OO modelling, it is also one of the most difficult to master. In particular, the mechanism of method redefinition is problematic and raises many conceptual design issues in the context of the class hierarchies.

The various proposed models of the concept of inheritance [Tai96] have undoubtedly affected its essence. It is the obscure uses of inheritance that raise alarms concerning its interpretation and validity. To date, the various interpretations are still subject of debate and the characterisation of good uses of inheritance is problematic. Clearly, the design process requires the application of skills and experience from the designers. When OO models present unconventional or suspect uses of inheritance, the reuse, the extensibility and the maintainability of such models are compromised. It should be noted that the advent of OO programming languages has also contributed to the disagreement on the correct use of inheritance. One possible approach to tackle such a problem is to reduce the risks for such suspect uses. To do so, guidelines also referred to as *recommendations* or *heuristics* [Fir95, Rie96, Rum96] have been proposed in order to identify and to express the “good uses” of inheritance. In general, heuristics appear as short textual description of the appropriate usage of the OO concepts. Although a heuristic may be conceptually understandable, the verification that an OO model satisfies it is difficult. Technically, depending on the nature of the heuristics, suitable verification methods do not always exist. The area of measurement techniques addresses such problems and is still actively researched. Ideally, OO design methods aim at providing techniques or principles for the evaluation of “goodness” or “badness” of an object model.

Assessing a design is difficult. Measurement science has suffered from criticisms concerning its usefulness [Bas&al95, Bou89, HarNit96, HitMon95a, Kow93]. Nevertheless, it is generally recognised that measurement techniques are beneficial for tackling issues during the software development life cycle. Assessment methods can be used for quantifying a particular design aspect. Most of the founded criticisms in the literature concern the correctness of the metrics themselves [Fen91]. The difficulty of acceptance of assessment techniques<sup>2</sup> from the developers’ community is due to the additional burden involved in putting a measurement programme in place. Also, unclear or non-meaningful feedback from the analysis of metrics results does not encourage the use of such techniques. However, it has been generally recognised that traditional metrics

---

<sup>2</sup> Assessment and measurement techniques will be interchangeably used but the latter term will imply the use of metrics as the underlying technique.

[Fen91] are not appropriate to the assessment of many aspects of object technology due to the fundamental differences. Recent experiments with novel set of metrics [Bri&al94, Hen96, Kem96] have therefore demonstrated the usefulness of the metrics in an OO context and emphasised the need for further research. Again, it should be noted that the fast moving industry of object technology has not favoured the adoption of measurement techniques during the design process. Often, designers still rely on experience and “feel” for the evaluation of the quality of the design. It is believed that the provision of adequate measurement tools will embody the designers’ experience and knowledge and thereby, will enable a smooth integration of measurement techniques as part of the crucial design process. To do so, the quantification of the level of goodness of an OO model necessitates a clear understanding of the recommended uses of the object concepts as well as the identification of the context in which unusual uses of the concepts may arise. Such issues can be addressed by heuristics and the derivation of appropriate metrics on the object model is expected to shed light on potential unseen complexities of the design.

Another important aspect of measurement techniques relates to the final phase of a measurement programme: *the analysis and interpretation phase* [Bou89, BriCuc98, Ebe92, Hen96, RosHya96]. In the current literature, this area has seldom been addressed although fundamental to the overall process. Usually, the derivation of metrics produces large data sets which require relevant analysis methods without which meaningful conclusions cannot be extracted. Often, graphical representations of the raw data sets facilitate the analysis process as unusual curves or charts may indicate potential problems. However, it is believed that such a process can be further enhanced in two ways:

- The use of various types of graphical representations. Often, metrics results are represented as bar charts or curves but many other types of representations may also be appropriate. The identification of characteristics of each may guide the process of interpretation to the desired conclusions on the design.
- The use of various functions for narrowing down large data sets facilitates the analysis process. Typically, when the conditions on which a problem appears have been identified, it is interesting to isolate only the metrics results concerned.

In order to tackle the problem of evaluation of quality of an object model, this research work envisages measurement techniques as the main instrument for design evaluation.

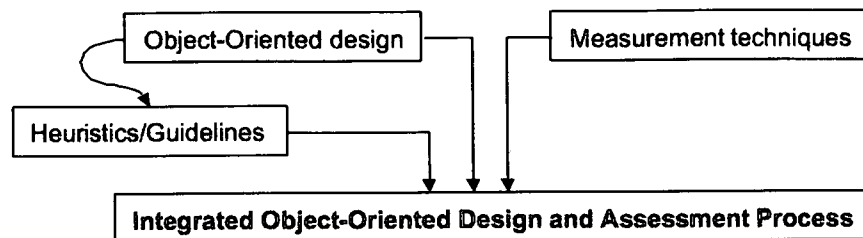


Figure 1.1: Objectives of the research work

Figure 1.1 depicts the objectives of the work. To date, object oriented design methods, design heuristics and measurement techniques form three separate areas of research. This thesis aims at reviewing the main design aspects and factors relevant to the problematic evaluation of internal quality factors of an object model. In particular, the case of inheritance is investigated. The definition of an appropriate measurement plan is presented and it is demonstrated how the use of metrics on sample object-oriented models sheds light on the complexities involved in the design of class hierarchies. This work identifies the fundamental conceptual and technical issues for the creation of such structures. In parallel, the experiments with measurement techniques contribute to the definition of a possible integration of these techniques within the design process.

The motivation of this work originates from the following facts:

1. The use of inheritance is desirable in software applications. OO methods have largely illustrated their benefits in a learning context [Boo91, Boo94, Emb92, Fir95, Gra94, HenEdw94, Mey88, Mey97, Rum91, Wil96]. Nevertheless, its various uses, sometimes contradictory, still generate debate amongst research and industry.
2. From a technical point of view, the control of the property inheritance scheme in class hierarchies is complex and difficult [AdaMo195, Bou89, Mey97, Sei96, Ste&al96, Tai96, Web95]. To date, the concept of inheritance seems to have lost its original meaning to comply with the requirement needs.
3. Emerging experiments [Bri96, BriCuc98, ChiKem94, Dum&al95, HarNit96, Kem96] and popularity for measurement techniques seem to indicate that they represent strong candidates for contributing to the design process [Avo94, BarSwi93, Bas&al94, Bas&al95, Bri&al95, Bri&al94, CheLu93, ChiKem91, Hen95, Hen96, Hit95, HitMon95b, LewSim98, Lew95a, LiHen93, LorKid94, RosHya96, Whi97]. In particular, there is a need for further investigation of assessment techniques for the inheritance concept.
4. Language designers have produced powerful and expressive features that manipulate inheritance in ways which are sometimes questionable. The modelling gap between fundamental design concepts and the features of programming languages still raises alarms on the conceptual validity of a design solution [ArmMit94, Bou89, McKMon93, PapLeJ97, Rie96, Sho&al93, Whi96a]. Designers expect a design to be reusable and maintainable; however, there are no methods that guarantee such criteria.

To illustrate the benefits of the use of measurement techniques, Figure 1.2 shows an example of typical expected metrics results. This result has been extracted from chapter 5 and the detailed analysis can be found there.

The use of measurement techniques for the assessment of an object-oriented design enables the discovery of unseen behaviour or unclear design situations. The derivation of appropriate metrics for the design ought to guide the designers to satisfactory indications or directions for improvement of the characteristics assessed. Thus, measurement techniques give opportunities to determine the level of goodness or badness of the design.

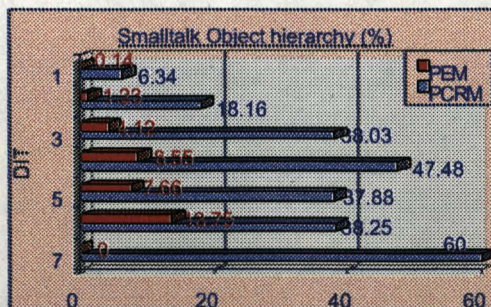


Figure 1.2: Measure of level of redefinition in the Smalltalk Object hierarchy

In Figure 1.2, the two bar charts represent the redefinition activity within the Smalltalk Object class hierarchy. The measures taken are for two types of method redefinition: extended methods (PEM) and replaced methods (PCRM). Clearly, the evolution of the redefinition activity down the levels of hierarchy can be seen. In particular, the amount of PCRM is much higher than the amount of PEM. Such a situation is unexpected and suggests further analysis of the peaks of the PCRM. The technique for metrics results analysis is detailed in section 3.4.

The thesis is organised into six chapters, plus “Glossary”, “Terminology”, “References” and “Appendix” sections. Chapter two provides a background literature review describing the relevant aspects of the concept of inheritance, the method redefinition mechanism, the area of heuristics and the measurement process. The definition of inheritance is reviewed and it is shown how its various interpretations affect the underlying property scheme. The identification of factors that can potentially produce inheritance misuses is sought. The description of the method redefinition mechanism clarifies the various ways of using the technique. Then, the investigation of heuristics and guidelines for object-oriented design sheds light on the correct ways of using OO concepts. Finally, to demonstrate the benefits of measurement techniques, the roles and the key practical aspects for building and using a measurement programme are highlighted.

Chapter 3 outlines the motivation for assessing the property inheritance scheme. Given the multiple interpretations and misuses of inheritance and the redefinition technique, a conceptual design problem referred to as the *multiple descendant redefinition* problem is identified. In order to

assess the inheritance characteristics of an object model, a methodological approach for class hierarchy assessment is described. Then, based on the GQM/MEDEA software quality model, a new set of metrics is proposed to measure redefinition in OO systems. Finally, a metrics interpretation framework is presented to tackle the lack of current assessment techniques concerning the extraction of meaningful feedback from metrics. In particular, an analysis and interpretation technique details the benefits of using various graphical representations to represent metrics results. It is shown how the discovery of unseen phenomena is facilitated and contributes to the extraction of satisfactory conclusions.

Chapter 4 presents a prototype of a metric collector tool that embodies the features for the derivation of the redefinition metrics. A brief analysis, design and architecture of the tool are given together with other implementation issues. This prototype tool enables the automatic derivation of the metric on class hierarchies, thus demonstrating the usability and applicability properties of the metrics. The user interfaces are described and implementation issues for the Smalltalk language are discussed.

Chapter 5 gives details of the experiments using the redefinition metrics. In particular, it is shown how the analysis of the results permits the detection of unexpected redefinition problems. Various graphical representations are investigated and the characteristics of each are outlined. In addition, a simple example of a detection technique is presented.

Finally, chapter 6 discusses and concludes the thesis. An integration model of the design process and assessment techniques is proposed and the potential benefits are presented. Further work envisions a promising future for the science of measurement within an object-oriented design process.



## 2. Background

*"The future has a way of arriving unannounced"* – George F. Will

*"Experience is the name so many people give to their mistakes"* – Oscar Wilde

Advances in object technology have helped many aspects of the design process in software development methodologies. Object-oriented methods aim at capturing and formalising the knowledge for designing good quality software. Although object concepts seem well understood and have proved beneficial, obtaining an acceptable and good object model is non-trivial. Designing software remains hard. For years, a considerable amount of research literature has discussed the concepts of object technology [Boo91, Boo94, Emb92, Fir95, Gra94, HenEdw94, Mey88, Mey97, Rum91, Wil96]. Using object-oriented technology for design reflects the natural desire of the industry and research community to manipulate concepts which seem appropriate for solving real-world problems. However, the *impedance mismatch* or *modelling gap* between the object concepts and the features of OO programming languages has recently changed the view of some authors of design methodology [Eli95, Liu96, Whi97]. *"Is C++ a high-level or low-level language? It depends how you use it!"* stated Coplien [Cop92]. Implementation considerations should be made in the design phase and not left to the programmer's own decision. This modelling gap has affected the design process in two ways:

- Considering both object concepts and the various implementations of the concepts in languages leads to many alternative choices for the object model, thereby making design decisions difficult.
- To keep the flexibility and "informality" of design activities for building various candidate object models to a single problem, assessment techniques have been studied and have shown to be promising as a design assessment aid towards the choice of the best suitable object model i.e. the most appropriate trade-off.

Software design is an art therefore it can be assimilated as a creative process. Meyer stated that in advanced software design there is no substitute for fresh thinking and creative insights [Boo94, Cha&a192, Col&a194, Gam&a195, Gra94, HenEdw94, Lew95b, Mey97, Rie96, Rum91]. Software methodology provides us with good advice from past experiences. Rather than a strict guide to design, methodologies propose flexible and general guidelines for software design. When good quality software is obtained, implicitly, this pre-supposes that the "goodness" of a design can be recognised from its "badness". From a designer's viewpoint, badness seems to appear easier to recognise because of the currently known pitfalls [Web95] or obstacles [AskBer92] occurring during software development.

The following list describes the generally accepted characteristics for “good” software applications:

- **Usability:** features of the software should meet the requirements and be usable.
- **Maintainability:** developed features should be as easy as possible to maintain with a minimum of disturbance.
- **Evolution:** related to maintainability and reusability, the software should be open and flexible enough to evolve with new requirements.
- **Reusability:** in a general sense, designers ought to reuse existing abstractions with minimum of effort.
- **Reliability or robustness:** software applications should work in various circumstances i.e. expected and unexpected situations should be tackled and the behaviour of the system should work in a deterministic manner. In the case of unpredictable events, recovery mechanisms should be provided.

Although most of the above criteria are desired when building applications, past experiences have shown that during design, there has to be a trade-off. The first reason for this comes from the fact that not all the criteria may be satisfied at the same time. The second reason is that the choice of the criteria to be satisfied mainly affects the overall cost of the development. Software engineering, which has existed for nearly four decades endeavours to bring solutions to this software dilemma. It is noticeable that problems that were qualified as complex in the past generally become more understood or solved with time.

In order to tackle the problem of assessment of an OO design, this background literature covers two main topics as follows:

1. **Inheritance and method redefinition:** section 2.1 presents the notion of inheritance and illustrates the problem of designing and identifying a correct class hierarchy. It is shown how inheritance shifts from its formal definition and can be interpreted differently in OO programming languages. As one of the main aspects of inheritance in a class hierarchy is the behavioural aspect, a detailed description of the important mechanism of method redefinition is given in section 2.2.
2. **Heuristics and assessment techniques:** section 2.3 explains how and why the problem of assessment of object models can be tackled by the technique of heuristics during the design process. Section 2.4 describes the area of assessment techniques and highlights its potential benefits for the improvement of the quality of OO designs. A software quality model presents the various aspects to be considered if a measurement plan is desired.

## 2.1. Inheritance and associated problems

### 2.1.1. Use of inheritance

*"Systems are not born into an empty world"* – Bertrand Meyer [Mey88]

The inheritance mechanism is one of the key features for the extensibility and reusability aspects of object-oriented systems [Boo94, CapLee93, Fus94, Gam&al95, HenEdw94, Mey88, Mey92, OOP93, Rum91, Sha92]. The concept of inheritance was introduced nearly 30 years ago in the Simula language [DahNyg66]. It has since become the core concept of the OO paradigm and one of the most controversial topics of research for the last decade.

Many researchers have shown that the use of inheritance in OO systems is still very low [HarNit96, Kem96]. It is suggested that the main reasons for this current state might be "the culture of the developer", the performance considerations, the complexity of its use and the amount of effort needed for maintenance and control of such systems. Inheritance has not been fully investigated. Class hierarchy design necessitates a great effort of creativity and the main difficulties lie in the fact that future additions of classes should be taken into account [Kem96, Rum96]. Whether those characteristics are predictable or not influence the shape and structure of the hierarchy [Fir95].

Recently, a variety of models of inheritance have been well described by Taivalaari [Tai96]. Although they offer a vast extent of expressiveness, all of the different mechanisms are still subject to conceptual design inconsistencies [ArmMit94, CapLee93, Fir95, Sei96]. In order to reuse the features of classes, designers face the problem of property (attribute and method) reuse [Dev96, KosVih92, Rum96] and method redefinition. The latter is a powerful mechanism that permits behavioural flexibility in a class hierarchy however, it can also introduce inconsistent design situations if wrongly used [Dev96, KosVih92, Mey88, Rum96, Sei96, Tai96].

Different languages allow different control structures and mechanisms to support reusability and extensibility. Conceptually, the idea of achieving reusability is not new. In any type of approach to a problem, the rule of thumb is *"not to re-invent the wheel"*. The term "reuse" has generated a lot of discussions within the research community as well as in industry. The promise of object technology lies, for a major part, in the reuse of the existing code. Code reuse takes its origin from the fact that a portion of code could be isolated and reused in another context. Thus, from a simplistic point of view, reusability is seen as code reuse. Programming is similar to any type of engineering process whereby factorisation and generalisation are necessary steps in order to obtain consistent and generic "modules". Lalonde and Pugh [LalPug91] claim that hierarchies are different structures depending on the notion of *subclassing*, *subtyping* or the *is\_a* relationship used when designing. Hierarchy design is always guided by rules or recommendations that are described in OO methods.

The validity of a class hierarchy is one of the most difficult tasks to assess. One can argue that a system is considered good when it is functionally correct. In such cases, the appropriate strategy to ensure the validity of the system is a rigorous testing strategy. This area of testing is beyond the scope of this thesis; however, testing and measurement techniques could act as complementary techniques. Evaluating the quality of a class hierarchy also concerns the evaluation of its structural and behavioural organisation. An approach to assess a class hierarchy for criteria such as reusability, extensibility and conformance can be tackled by measurement techniques [Bas&al95, Bri&al95, ChiKem94, Dum&al95, Hen96, LorKid94]. Although there are common requirements and expected features of class hierarchies [AdaMol95, Mey88, Tai96], the variety of inheritance models lead to different class organisations due to an emphasis on particular criteria to be achieved. Thus, the existence of different approaches to class hierarchy evaluation. Compilers already encompass technology to detect design errors such as type checking in strongly typed languages such as C++. The principle of *substitutability* or *conformance* i.e. the type of a subclass should conform to the type of its parent(s), is then ensured.

Given that the aims of this thesis are to assess a particular aspect of inheritance i.e. the method redefinition principle, sections 2.1.5 and 2.1.6 cover the property inheritance mechanism. As a class hierarchy is the main structural organisation using the full potential of the inheritance relationship, emphasis will be put on the issues involved in designing such an architecture as well as the possible ways of evaluating its design quality factors. However, it is important to consider different aspects of inheritance which are necessary for better assessment. In particular, attention will be paid to the "*inheritance scoping control*", as it is the core mechanism permitting the expected benefits of OO technology. Similarly, it will be interesting to look at the recognised design problems associated with the use of inheritance. If it is possible to clearly identify typical problems, it will be easier to detect and correct them.

In the next section, the study is mainly based on current OO programming language constructions although not losing sight of a more theoretical view of the inheritance concept. The reason for this lies in the variety of possible constructions offered by languages. Although being design considerations, current OO methods do not encompass a description of those constructions as they are often language specific. For instance, in Eiffel, it is possible to specify invariants, assertions and the list of client classes that are allowed to use the class properties. This creates the semantic modelling gap between OO methods and OO programming languages.

### 2.1.2. Class hierarchy organisation

The construction of class hierarchies still remains a problem due to the constraints involved and the required criteria. The inheritance relationship is used in order to strongly couple classes in a parent-child scheme. Although property inheritance constitutes a powerful mechanism for achieving reusability and flexibility, it can also introduce inconsistencies in design that infringe the

essence of inheritance. When developing an OO application, it is common to use libraries of classes which provide general functionalities to a specific domain. Usually, a class library, organised as a tree hierarchy, becomes part of the system developed. The main function of class hierarchies is to provide the developer with an organised set of reusable and extensible classes. For instance, all programming languages encompass such libraries for managing widgets, networks, collections, etc. In the Borland™ C++ integrated development environment, the hierarchy is known as the *Object Windows Library* (OWL) and provides the developer with the windows management API. Microsoft™ and Sun Microsystems, Inc. have equivalent libraries respectively called the *Microsoft Foundation Class* (MFC) and the *Java Foundation Class (JFC)/SWING* [Sun99].

In a class hierarchy, the classes newly added to a hierarchy extend and inherit from the classes present in the hierarchy. The Smalltalk class hierarchy provides a single-root class called *Object* and does not support multiple inheritance. Single inheritance simplifies the architecture of a system and makes the maintenance easier, whereas the use of multiple inheritance involves additional problems such as name space conflicts. Although it is possible to find equivalent solutions to single inheritance structures, the benefits of code reuse may be compromised. Inheritance in OO systems provides a feature dispatching mechanism that allows the sharing and selection of the code.

In many occasions, real-world objects have common behaviours but are realised in different ways. For example, consider the two classes *Bag* and *OrderedCollection*, which are both structures for storing elements. A bag contains elements with no particular order as opposed to an ordered collection of elements which is indexed on a key. Both classes have the same behavioural semantics of adding elements in the structure but in the case of an ordered collection, a key must be provided in order to record the position of the element in the structure. Therefore, the implementations are different but the interfaces can be the same.

Unfortunately, class organisation is problematic as many viable design solutions may be discovered depending on how the notion of inheritance is used. The following sections describe three main categories of inheritance uses that raise the problem of correctness and appropriateness of each.

### 2.1.3. Subclassing, subtyping or specialising

Different class hierarchy organisations can be designed depending on the model of inheritance used. Taivalaari [Tai96] showed that three completely different tree hierarchies can be drawn depending on the relationship used for design.

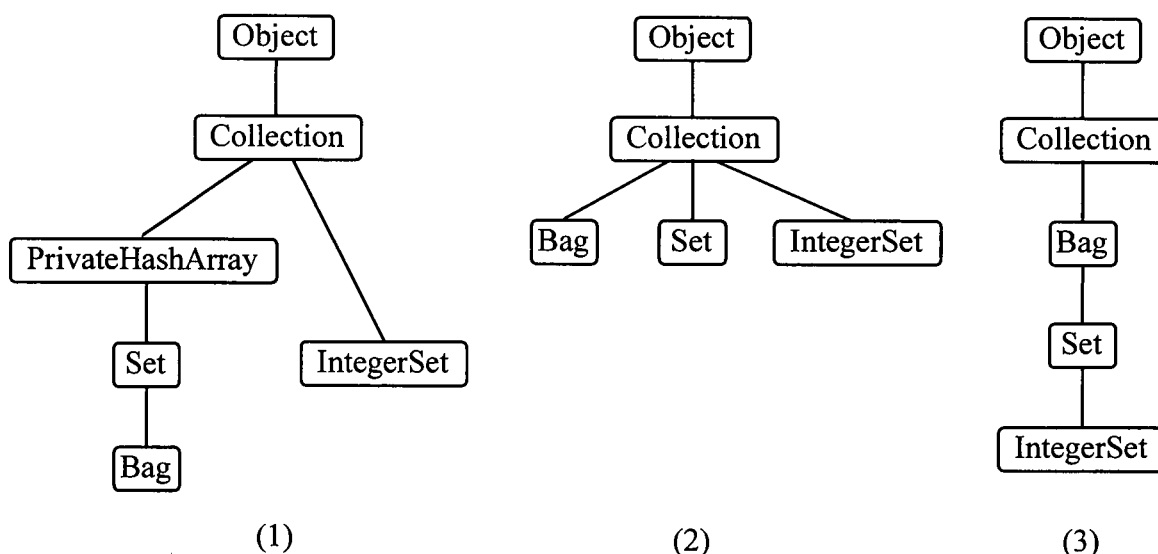


Figure 2.1: Subclassing (1), subtyping (2) and specialisation (3) hierarchies

In Figure 2.1, the three possible hierarchy organisations are shown. Each of them represents a possible use of inheritance for modelling the different kind of `Collection` classes.

Subclassing refers to an implementation mechanism where the purpose is to share code. For example, the Smalltalk class hierarchy has been criticised for its non-conventional use of inheritance i.e. *implementation inheritance*. When the addition of new classes in the hierarchy (i.e. subclassing) is done because the parent class holds the required services without consideration for other services which might not apply to the new class' instances, the new class does not conform to its parent class. Thus, as the new class inherits all unwanted services from its parent class, incorrect calls can take place, and potential exceptions can be raised. In all weak-typed languages, this kind of inheritance is possible to implement. Implementation inheritance as discussed by Meyer (see inheritance taxonomy<sup>3</sup> detailed in [Mey97]) can be referred to as a legitimate case of use of inheritance. Meyer emphasised that implementation inheritance is conceptually valid as long as the subclass still conforms to the parent class. Recall that one of the criticisms concerning implementation inheritance relates to the fact that a subclass would conform to only some of the parent's properties, ignoring the remaining although inheriting them. In a better design situation, the parent would have cancelled the unnecessary properties for its children. However, assuming that a type equals a class, such non-conformance of classes in a hierarchy can be detected at compile time.

Subtyping refers to a substitutability relationship between a subclass and its parent class. This directly relates to the type of OO programming languages. Languages dictate the development spirit as they belong to two main categories: weakly typed and strongly typed systems. Strongly typed languages claim that the development of more reliable applications is possible while weakly

<sup>3</sup> Note that the described categories represent only a subset of Meyer's inheritance taxonomy [Mey97], relevant to the analysis.

typed languages have a high productivity rate with not much overhead. The type-check is done at compile-time for the former while an exception would be raised at run-time for the latter. Until now, the commercial market has been mainly interested in strong-typed languages such as C++ and Ada. However, interpreted languages such as Smalltalk usually offer a rapid development environment where software applications can be quickly prototyped and tested. One noticeable difference between the two categories of languages relates to the inheritance hierarchy structure. Although two hierarchies may be different, they may satisfy the same requirements. This emphasises the fact that the goodness of a class model (e.g. class hierarchy) is difficult to define as well as difficult to realise.

Specialisation inheritance respects the conformance rule. A child class *is\_a* particular type of the parent class, therefore a specialisation of the parent. Another way to describe this mechanism is to consider a subclass as a subset of its parent classes whereby all features of the parent apply to all its heirs. Conceptually, specialisation inheritance permits a clear categorisation of objects regarding their intrinsic properties; therefore it encourages the use of abstraction.

Note that the root class, in a single-rooted inheritance tree must be the most abstracted class in the hierarchy. Smalltalk's root class<sup>4</sup> encompasses all the generic behaviour inherited by all the subclasses. A single-rooted approach for a class hierarchy incurs some problems for the management of the classes. For example, when developing an application with Smalltalk, the library classes and the application classes are built within the same class hierarchy. This non-separation of provided or newly built classes makes the release of an application difficult.

These different categories of class hierarchies may impose severe restrictions on some aspects of the future development of the hierarchy. More research is necessary in this area in order to clearly identify all possible effects and problems incurred by the use of a particular category. It is, however, possible to evaluate the "goodness" of a class hierarchy regarding two crucial aspects: usability and extensibility. This is described in the next section.

#### 2.1.4. Usability and extensibility

Two main quality factors are the *usability* and *extensibility* of class hierarchies. The notion of extensibility refers to the capability of adding new features to a class or new classes to an existing class library. New classes are seen as specialised versions of their parent classes. An inheritance relationship indicates a strong form of coupling between the classes where common behaviours are shared. This kind of use relates to a functional-orientated approach whereby the use of a class is

---

<sup>4</sup> Many recent discussions from the X3J20 committee for Smalltalk [X3J96] standardisation has raised the question of having a class hierarchy inherit from nil instead of the Object class. This would enable the creation of many class hierarchies rather than a single-rooted hierarchy.

accepted when the required functionality exists regardless of the conceptual correctness of the classes.

The major difficulty when using a class hierarchy depends on the level of depth of the tree. The deeper the level, the more difficult the understanding and use of the classes. This is where the concept of inheritance is paradoxical in the sense that, in theory a class hierarchy should be deeper because it increases the general level of abstraction, but in practice it rapidly becomes difficult for humans to master deeper levels in the hierarchy. Therefore, there is a large burden for the user if attention is not paid to building hierarchies where child classes conform to parent class(es). Although the level of difficulty can be defined differently among designers, current commercial class hierarchies are not straightforward to approach and this raises the need for further research in making efficient use of complex hierarchies. For instance, suppose that it is required to extend a particular branch of a hierarchy which is already deep (Riel [Rie96] considers a level deep when it reaches the magic number seven), it becomes difficult to understand behaviour of each class in the branch.

With the concept of a *class contract* [Mey88, Ste&al96], emphasis is put on the specification of the interfaces of the class. If each class encompasses a high number of publicly available methods which are inherited down the branch, the final concrete class from which extension is planned becomes difficult to understand. Indeed, the first step to extend the hierarchy is to localise the correct class from which it is relevant to subclass the new class to be added. A quick look at the class names in a particular branch should already pinpoint interesting classes to reuse. A simple approach is to "look-up" classes higher in the hierarchy in a bottom-up fashion. Briefly, starting from the closest parent from which a derivation is desired, it is possible to scrutinise the class in order to find desired abstractions. Thereafter, the same approach for higher classes in the hierarchy can be taken. In the case of multiple inheritance, a multiple descendant path has to be studied with attention to possible conflicts such as the name space conflicts from repeated inheritance [Mey88]. Whenever a new class is introduced in a hierarchy, it should conform to all ancestor classes. Without tool aids such as class hierarchy browsers in IDEs, it is difficult to understand classes from an existing hierarchy. If CASE tools are used, the designers' task becomes easier because of the graphical representation of an object model.

Other problems of class reuse and extensibility relate to a psychological issue. One of the heuristics provided by Riel [Rie96] states that the design of a branch of a hierarchy should be given to a single architect designer. This comes from the fact that developers tend to implement their own versions of programming code as soon as there is a suspicion of possible unreliability of existing code. In many cases, it appears that re-implementing code is much faster than trying to understand and modify what has previously been done. Indeed, this is not recommended, but it happens for many reasons:



- Programming practices of each developer: everyone has his own style of programming e.g. syntactical language construction or presentation, algorithmic preferences, etc.
- No available documentation explaining previous class behaviour and semantics.
- Complex dependencies between classes: if classes are strongly coupled, it is very difficult to understand the general behaviour of one single reference to an interface. Also, this refers to the problem of undesired side-effects generated by method dependencies.

Class addition is one way of extending class hierarchies. Another possibility of extension can be done within an existing class itself. Typically, the extension of a class interface broadens the behaviour of the class. The higher a class is in the hierarchy the more abstract it is, which means that the behaviour must also be abstract enough so that it will be relevant to all subclasses, otherwise the conformance rule is broken. For this reason, deletion or modification of the behaviour of an existing class is highly critical as other client classes might rely on the deleted behaviour or expect a different behaviour. These class and hierarchy management issues are studied in the schema evolution research area for databases. Further details can be found in [BanKim87, Bar&al93, Ber91, Cas93, CheLee96a, Dic95, Gib90].

The support for reusability and extensibility through inheritance is different across object-oriented programming languages. It relies on the type of inheritance scheme used (see section 2.1.3). Consider a class A in the Smalltalk hierarchy:

```

Object subclass: #A
instanceVariableNames: "
classVariableNames: "
poolDictionaries: "

```

Class A is declared as a subclass of the root class **Object**, therefore A inherits all variables and methods that the **Object** class holds. There is code reuse as soon as the subclass A uses inherited behaviour. In Smalltalk, there is no declarative construction which forbids a subclass to inherit from a parent's property. With the various constructions allowed in programming languages, it is possible to introduce conceptual inconsistencies particularly when using inherited redefined methods. The incorrect use of redefinition leads towards incorrect classification and furthermore to an incorrect behavioural inheritance [ArmMit94, Hen96, Mey97, Rum96].

Given the possible uses of inheritance in class hierarchies described above, the next section presents a formal definition of inheritance and highlights the implicit property inheritance scheme suggested by the definition. It is precisely the way the property inheritance scheme is used that enable the designers to produce conceptually orthogonal class hierarchies.

### 2.1.5. Property inheritance scheme definition

Inheritance is the main mechanism which supports the realisation of criteria such as reusability and flexibility [Hen94, New&a196]. An addition of a class to an existing class hierarchy specialises a branch of the tree, thereby extending it. By inheriting features from ancestor classes, reusability is also achieved. However, there exist many models of inheritance and the correct application of any model is debatable [LiHen93, Sei96]. The formal definition of inheritance is characterised as follows [BraCoo90, Tai96]:

$$(1) \quad C = P \oplus \Delta C$$

where a new class  $C$  is shown as a combination ( $\oplus$ ) of a set of properties inherited from an existing class  $P$  and the new properties ( $\Delta$ ) which make  $C$  a specialised version of  $P$ . In this equation, the relation superclass/subclass is assumed to be transitive, therefore  $P$  includes all cumulated properties from its own parents. However, the inheritance scheme of properties from parent class to child class is open to many interpretations. Taivalsaari [Tai96] explained that  $P$  represents the properties inherited from an existing object or class where, in fact,  $C$  is able to inherit from many classes either in the same descendant branch or multiple branches if in a multiple-inheritance situation. It is generally accepted that the deeper a class is in a hierarchy, the more difficult the control of inheritance becomes. Therefore, leaf classes are more subject to bad design than their parents are.

To illustrate how the properties are inherited in equation (1) according to the definition of inheritance, the set of properties of a subclass  $SubCls$  of a class  $Cls$  becomes:

$$(2) \quad SubCls = Properties(Cls) \oplus Properties(SubCls)$$

where

$SubCls < Cls$  i.e.  $SubCls$  is\_a subclass of  $Cls$ ,

$Properties(class) = \{ inst \mid inst \in \langle Attributes \rangle, mth \mid mth \in \langle Methods \rangle \}$

$Properties(class)$  is the set of attributes and methods of a class i.e.  $\langle Attributes \rangle$  and  $\langle Methods \rangle$  respectively refers to the set of possible instance variables and the list of methods in the class.

Introducing the origin of properties in (2) gives:

$$(3) \quad SubCls = Properties_{inherited}(SubCls) \oplus Properties(SubCls)$$

where  $Properties_{inherited}(SubCls) = \{ x \mid x \in Properties(Cls), x \text{ is publicly available to } SubCls \}$ ,

From (2) and (3), a subclass  $SubCls$  is a combination of its inherited properties and its currently defined ones. (3) introduces properties overlapping in the definition when reuse of properties is achieved.

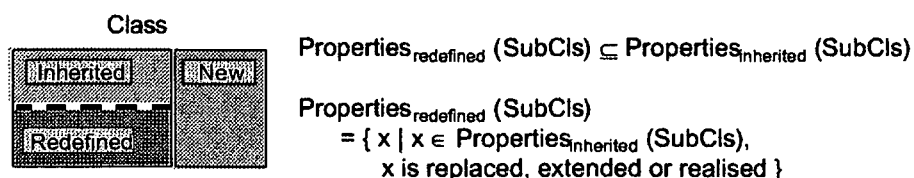


Figure 2.2: Class properties

$Properties_{\text{redefined}}(\text{SubCls})$  are the (inherited) redefined properties as opposed to  $Properties_{\text{inherited}}(\text{SubCls})$  which is a superset including the ones accessible and used without modification. Because of the variety of possible modifications to a property such as complete redefinition, extension or realisation, there is a possible source of incompatibility between a class and its subclass. As stated by Taivalsaari, inheritance use does not guarantee a conceptual specialisation intention. The mechanism of redefinition has been criticised [ArmMit94, Fir95, KosVih92, Mey88, Rum96, Tai96] for not bearing any kind of semantic relationship with its initial implementation, especially when the method is completely overridden. Unfortunately, the *inheritance "scoping" control*<sup>5</sup> facility does not prevent this conceptually inconsistent situation. Indeed, a non-strict *is\_a* policy is more likely to introduce unsubstitutable classes and is used either for convenience reasons or because it *uses\_a* parent class property.

This section reviewed the formal definition of inheritance and showed the implications of the definition with regard to the property inheritance scheme. It becomes clear that property inheritance is a key aspect to the assessment of class hierarchies. The next section describes the property ownership transfer and the consequences on the design.

#### 2.1.6. Property ownership transfer

As seen in the previous section, the property inheritance scheme states that properties of a parent class should be inherited by all its heirs whatever the level in the hierarchy. In a child class, visibility and accessibility of a property is defined in the parent class. This means the child class is then able to change the property values i.e. **public** inheritance of properties implies a property ownership transfer from the parent class to the child class (Figure 2.3). Due to application requirements, e.g. business rules, restrictions have been added to this notion of inheritance. Not all properties of a parent class can be inherited by its subclasses. The representation of a real-world entity by an object often necessitates hiding some of its properties from other interacting objects i.e. *encapsulation*. This facility permits an object to manage internal properties for its own purpose. In OO programming languages, attributes declared as **private** can only be accessed within the class where it has been defined. Private attributes are not inherited by heir classes. The

<sup>5</sup> The process of declaring appropriate modifiers to a class, an attribute or a method will be referred to as the *inheritance scoping control* facility.

main variants of property inheritance features of four OO programming languages are described below. It is important to note that all possible features allowed by programming languages are subject to design problems when not used correctly.

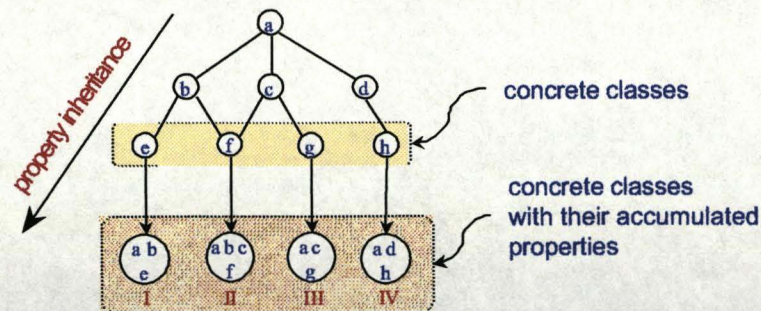


Figure 2.3: Transfer of property ownership in an inheritance hierarchy

In OO programming languages, the transfer of property ownership (Figure 2.3) is realised by the application of property modifiers to the property. The next section presents various encapsulation schemes offered by programming languages and illustrates their fundamental differences.

#### 2.1.7. Encapsulation: visibility and accessibility of properties

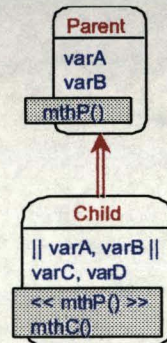


Figure 2.4: Example of transfer of property ownership

In OO languages such as C++, Eiffel or Java, there are syntactic control declarations which allow the control of the scope of the inheritance of the properties from a parent class (Figure 2.3, Figure 2.4). Various control schemes are available depending on the language. Property modifiers can be applied at class, variable and method level. Although there are exceptions, in most languages the inheritance of properties is done in a top-to-bottom direction. It is the parent(s) class(es) which define the properties to be inherited by its heir classes (Figure 2.4). The main three basic property modifiers are **public**, **protected** and **private** [Str90]. In the previous paragraph, **public** and **private** were presented. When the **protected** modifier is applied to a property of a parent class, only its subclasses are able to access the property. This mechanism restricts the visibility and accessibility of a property to descendant classes in a particular branch of the hierarchy.

Other types of modifiers exist depending on the language. For example, in Eiffel, a parent class is able to define a subset of its subclasses which is going to inherit a particular property as opposed to all of them. Stopping the inheritance of properties as described in section 2.1.8 is conceptually questionable as it breaks the transitivity mechanism of inheritance. Although valid reasons exist for the presence of such modifiers e.g. optimisation, standardisation and security, the mechanism appears as a language feature issue which conceptually affects the quality of the design.

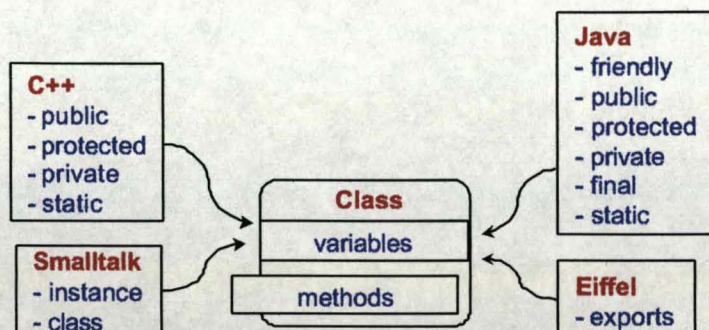


Figure 2.5: Property modifiers in OO programming languages<sup>6</sup>

One of the aims of the authors when developing the Java programming language was to provide a language for which developers would feel familiar with. For this reason, Java syntax [Tea&al96] is close to the well-established C language (Figure 2.5). Most of the complexity of the C++ was removed although retaining the main features. Java claims to improve the flexibility and maintainability of programs. Note that other modifiers may exist for the illustrated languages, however, they are not relevant for the purpose of this document.

The following description covers the main arguable modifiers in Java. In addition to these modifiers, it is possible to define *packages*, which are viewed as self-contained modules. Syntactically, a Java class declaration is of the form:

*modifiers class newClass*

Modifiers does not actually affect the class itself but determine how the class will be handled in case of addition of new classes or features to classes. Modifiers, in Java, are of different types: *friendly*, *public*, *private*, *protected*, *final* and *abstract*. In C++, when the *friendly* or *public* modifiers are applied to a class, other client classes have a full access to the properties of the server class. The only difference is that *friendly* provides access to classes in the same package i.e. group of classes.

The semantics of some modifiers are controversial because of the consequences of their use. It becomes difficult to control the whole list of properties together with restrictions imposed by the modifiers at each level of the hierarchy. In addition, side-effects are easily introduced when classes are complex. For example, encapsulation can be violated when attributes of a class are declared as

<sup>6</sup> The figure only shows the relevant property modifiers of four OO programming languages.

public, giving direct access to subclasses. The correct choice of property modifiers is an important task when an inheritance relationship is used but also remains one of the arduous design issues. Some errors can be statically checked by the compiler, or dynamically in the case of an interpreter. Unfortunately, this has long been seen as an implementation issue. It is arguable that such problems are directly dependent on the architecture and design adopted, therefore language features influence and affect the design. Only recently, such considerations have been described as part of the analysis and design methodology [Col&a194, Mey97, Whi97].

The next sections will show how the concept of inheritance shifts from its definition and why the design of class hierarchies becomes even harder with the constraints imposed by new types of information systems.

### 2.1.8. Consequences of encapsulation on the inheritance scoping control

#### **Controlling the property inheritance scheme**

In current software development methodology, little has been described about ways of controlling the property inheritance scheme. In practice, in a commercial class library, the amount of inherited properties in the leaf classes is usually high. In consequence, tracking back the different uses and definition applied to properties in ancestor classes is not straightforward. Often, it is assumed that properties and behaviours have consistent semantics. The knowledge of the history of inherited properties is crucial when considering the addition of new classes to an existing class hierarchy. Possible design errors concerning the conformity of a class to the parent(s) class(es) are then reduced. It is noticeable how inheritance is still not generally used or accepted in industry. Cartwright [Car98] stated that only “experts”, i.e. persons who know how to control and maintain complex inheritance structures, were doing so.

When considering the essence of inheritance and its uses [Tai96], designers are facing the dilemma of using powerful features of languages without being able to completely control the effect of their use [ArmMit94] e.g. Java language. It can be argued that the control over property inheritance only adds an additional workload for the designer, as there is no recognised common standard set of modifiers (Figure 2.5). Instead, each programming language has its own syntactic constructions. For example, in Smalltalk Express<sup>7</sup> there is no equivalent method modifier for the `private`<sup>8</sup> keyword in C++ or Java. Any method in a class can access any other method declared in another class. Therefore, at method level, Smalltalk provides the designer with fewer features to ensure information hiding. Instead, programmers need to keep in the “spirit of OO” and not infringe the rules, although this is possible. Often, theoretical and conceptual issues are ignored in favour of

---

<sup>7</sup>In this thesis, Smalltalk Express™ designates the version based on Smalltalk/V® Win16 and WindowBuilder® Pro/V provided by ObjectShare®, a Division of ParcPlace, <http://www.objectshare.com>

<sup>8</sup> In Smalltalk, all instance variables are defined as private whereas instance methods are publicly inherited.

pragmatic solutions [Tai96]. This situation has been generally recognised as an arguable use of inheritance as prediction and extension of an existing class hierarchy becomes difficult and un-maintainable. Clearly, there is a need for additional control of the property inheritance scheme.

### **Abstracting for controlling inheritance**

Conceptually, classification techniques imply the existence of a category of classes with similarities from a structural and behavioural viewpoint. Therefore, it is sensible to have such a property inheritance scheme in order to cover a wide range of real-world problems. If a class holds methods which are to be inherited by some branches and not others, it might suggest a classic design problem where the parent class represents more than one concept, therefore containing methods which might not apply to all of its subclasses. On the contrary, it is recommended to use *abstract methods* (also called *deferred* in the Eiffel terminology) in a class where only the interface of the methods is provided and all subclasses are forced to give their own implementation. This type of inheritance is called *reification* inheritance. In such cases, methods in subclasses of the same class usually have different implementations i.e. *polymorphic* methods. In Smalltalk, declaring a method as abstract is not done via a modifier. Instead, the body of the abstract method contains the `implementedBySubclass` message which has the same effect (see example below).

```
Object subclass: #Test
```

```
    instanceVariableNames: "
```

```
    classVariableNames: "
```

```
    poolDictionaries: "
```

```
    Test instance methods
```

```
    realisedMethod
```

```
        ^self implementedBySubclass
```

For leaf classes, the immediate advantage is to reuse and extend the inherited properties. Often seen incorrectly as a simple code reuse mechanism, abstraction is a conceptual technique permitting the extraction of similarities from objects to form new coherent abstractions. Where a class contains one or more abstract methods, the class is referred to as an *abstract class*. By consequence, instantiation of an abstract class is prohibited. Introduction of abstract classes in a hierarchy is recommended. However, deep class hierarchies are still difficult to manipulate due to the many levels of depth. Often, this results in cases of ignored inheritance, especially when considering incremental development of classes. Nevertheless, it is generally recognised that the support for adequate documentation and tools reduces the risk of unusual inheritance situations.

### Case of multiple Inheritance

In OO languages that support multiple inheritance such as Eiffel, the publicly declared properties of all the parent classes are inherited by the subclass. Although the concept is sufficiently expressive to represent some categories of problem, the use of multiple inheritance generates obscure design problems concerning the property inheritance scheme. One of the most studied problems concerns the *name spacing* issue. When a subclass inherits from two parents (or more), all inherited properties should be accessible by the child class. If the parent classes contain properties with the same name, a conflict has to be resolved and the subclass has to decide which of the properties to inherit. In some development environments, the compiler statically checks for such problems and a default inheritance scheme may be provided when potential conflicts arise. Consider two base classes LIST and ARRAY which both define two features: print and show. With Eiffel, it is necessary to use the renaming mechanism to prevent name clashes.

```

class FIXED_LIST [T] export ...
inherit
  LIST [T] rename print as printList, show as showList;
  ARRAY [T] rename print as printArray, show as showPrint
feature
  ... specific features of linked-size lists ...
end -- class FIXED_LIST

```

Figure 2.6 illustrates another classic example of use of multiple inheritance. In a class library, the Stream branch provides a framework for managing data structures, input and output functionalities, sequential and random accesses. Intuitively, a ReadWriteStream class would make use of multiple inheritance and inherit from both the ReadStream and WriteStream classes. Then, a FileStream inherits from the ReadWriteStream, thereby all its parent's properties.

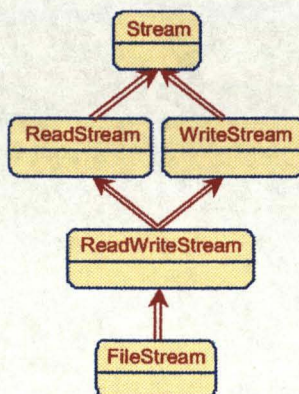


Figure 2.6: Stream hierarchy with multiple inheritance

Graphically represented in Figure 2.7, a new added subclass cumulates all properties of all its ancestor classes along the different branches.



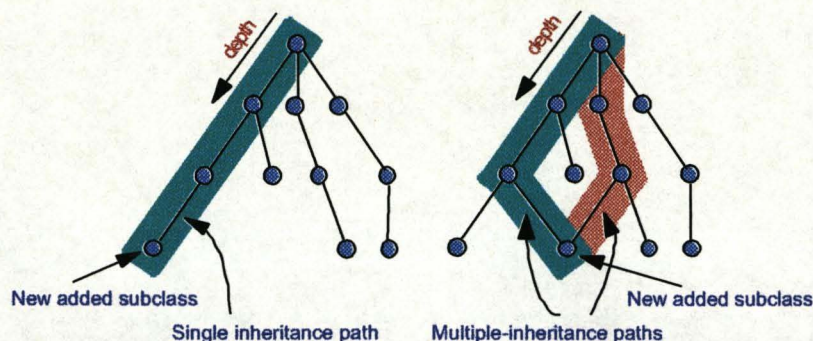


Figure 2.7: Traversal paths for single and multiple inheritance

Therefore, name space conflicts arise not only from the direct parent classes to the child class but from all ancestor classes.

In general, the levels of depth in class hierarchies affect the control of inheritance. Paradoxically, the abstraction technique promotes such a situation, thereby making the design of class hierarchies problematic. The next section illustrates the most common inheritance design mistakes. This is intended to present the underlying design issues and the recognition of good design practices.

#### 2.1.9. Common inheritance design mistakes

Over the past decade, many authors have presented cases of misuse of inheritance. Most of them argue the conceptual validity of non-conventional ways of implementing the inheritance relationship [AdaMol95, ArmMit94, Fir95, KosVih92, LalPug91, PapLeJ97, Web95, Wil96]. The main reasons given are that they affected one of the criteria such as maintainability, reusability or flexibility of the design. In most cases, the conclusion was that the arguable inheritance case presented impacts the overall cost of the development in terms of future evolution of the design. Rather than an exhaustive list of inheritance design mistakes, this section describes the main example problems and highlights the design attributes which are of interest in an assessment perspective. Also, it helps at recognising the classic design pitfalls for the identification of problems during a measurement programme.

One of the major problems in software development is, for any designer, to keep in mind all possible dependencies between components in the architecture. Meyer stated that modules should be understandable by themselves. If a component requires the knowledge of other information in other modules, it clearly shows that they are dependent on each other. Therefore, the change of one component might also require the change of the other i.e. they are dependent on each other. Although not recommended, as modules or objects rely on each other for communicating information, dependency or coupling exists. The issue is to control it. To help the designer, tool support is clearly desired [Bri96].

Another example of design issues concerns the paradox between what could be understood as an optimisation task and design tricks or tips. When a class relies on information given by another class, messages are sent back and forth according to the classic client-server model. An alternative design choice would be to make the information available in the original class, so that no messages are exchanged between objects. The reason suggested for such a choice is the possible gain in performance and context switching. This is a wrong design choice as the eventual benefit depends only on the internal architecture and algorithm of the compiler or interpreter. In addition, the original class might no longer constitute a single abstraction and possible duplication of information is likely to happen. Clearly, the design results in code of obscure quality. Often, common design inheritance mistakes are mainly due to the side-effects produced during incremental refinement and development of the classes. Examples of common design mistakes, identified in many OO methodologies [HenEdw94, Mey88, Rum91, Web95], are illustrated below.

- **Breaking encapsulation:** when a child class inherits from a parent class, the child class has direct access to all inherited properties including instance variables of its parent. Amongst other use of inherited properties, an instance of the child class is able to directly manipulate the value of an inherited attribute. As recommended by any OO method, access to a private instance variable should always be done by accessor, getter and setter functions.
- **Concept and implementation:** tree hierarchies have been widely used to mainly represent four abstraction principles:
  - \* Generalisation/Specialisation.
  - \* Aggregation/Decomposition.
  - \* Classification/Instantiation.
  - \* Grouping/Ungrouping.

The inheritance relationship definition validates the first case only. Often the *is\_a* relationship has been mistakenly used instead of the *has\_a* relationship (aggregate components) or *is\_implemented\_using* relationship (behavioural reuse facility). Although the representation as a tree hierarchy is conceptually valid, the relationship between the classes is fundamentally different. Consider the example below where a **STACK** class is declared as a subclass of the **LIST** class.

```
Object subclass: #LIST
  instanceVariableNames: 'listOfElement'
  classVariableNames: "
  poolDictionaries: "

STACK subclass: #LIST
  instanceVariableNames: 'top bottom currentPointer '
```

**classVariableNames:** "

**poolDictionaries:** "

A better design alternative defines the LIST class as an aggregate of the STACK class:

Object **subclass:** #STACK

**instanceVariableNames:** 'top bottom currentPointer listOfElement'

**classVariableNames:** "

**poolDictionaries:** " !

STACK instance methods

initialise

listOfElement := LIST new.

In his taxonomy of inheritance, Meyer [Mey97] refers to the first example as *facility inheritance*. He argued that this solution is perfectly viable and conceptually acceptable if all the behaviour provided by the LIST class can be applied to the instances of the STACK class.

Meyer identified two forms of facility inheritance:

- \* *Constant inheritance*: in which the parent yields constant attributes and shared objects.
  - \* *Operation inheritance*: in which it yields behaviour.
- **Class coupling generates dependencies**: any type of coupling between classes implies class dependencies. Lakos [Lak96] mentioned that for compiled languages, "*a component y depends on a component x if x is needed to compile or link y*". Many forms of coupling exist [HitMon95b] and sometimes, they generate hidden side-effects problems. For example, in the Lisp-based ITASCA™ Distributed Object Database Management System [Ibe94], the declaration of a class and its attributes has the following syntax<sup>9</sup>:

**(def-class DEPARTMENT**

**:document** "Department class" ;; comment about the class

**:superclasses** (ROOT) ;; parent class

**:abstract** NIL)

**(change-attribute 'DEPARTMENT 'Group :classp NIL**

**:document** "Instance variable Group"

**:inherit-from** NIL

**:composite** T

**:dependent** T

**:domain** '(set-of COMPUTING-GROUP)

**:init** NIL)

---

<sup>9</sup> ITASCA™ API is based on the Allegro CLOS language, Franz®, Inc.

In the above example, an instance variable named `Group`, of the `DEPARTMENT` class is of type `COMPUTING-GROUP`. The `COMPUTING-GROUP` class is declared as dependent aggregate (`:composite` keyword) of the `DEPARTMENT` class. In other words, all component aggregates (instances of `COMPUTING-GROUP` class) depend on their container part (instances of `DEPARTMENT` class). By consequence, a deletion of an instance of the container implicitly deletes the aggregate objects as well. This dependency mechanism is indeed dangerous if the contained objects should exist independently of the container objects.

Coupling can be categorised in three groups [HitMon95a]:

- \* **instance variable relationship:** in a client-server model:

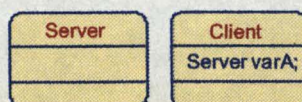


Figure 2.8: Coupling with instance variable

In Figure 2.8, the simplest form of coupling is done in declaring an instance variable: `Server varA`; in the `Client` class i.e. aggregation.

- \* **behavioural relationship:**

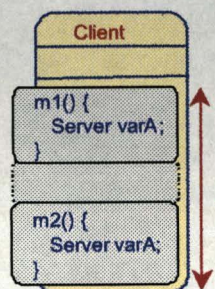


Figure 2.9: Coupling with method

In Figure 2.9, methods declare local variables of a particular class type. Although the scope of the local variables lasts only during the execution of the method, a coupling is nevertheless established.

A variant of the behavioural relationship is realised through the method signature:

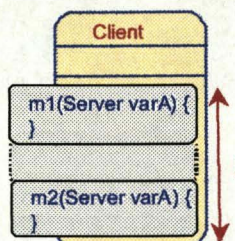


Figure 2.10: Coupling with method signature

In Figure 2.10, the coupling is realised in the declaration of the passing parameters. In order for the method to understand the argument types, the types are also declared within the method's signature.

\* **inheritance :**



Figure 2.11: Coupling with inheritance

In Figure 2.11, when a class inherits from a parent class, it also inherits all the publicly declared properties. This type of coupling is qualified as *strong coupling* as opposed to *weak coupling*.

- **Classification or objectification:** the problem of finding the best classes is still one of the major problems of OOD. Many methods propose an object-centred view to start off the design and apply abstraction wherever needed in order to extract potential classes. Alternative choices are always possible and the decision depends on the context and the specifications of the problem. For instance, there is sometimes hesitation in choosing between different constructs such as the use of an attribute or a class. Consider an **ENGINE** class modelled as follows:

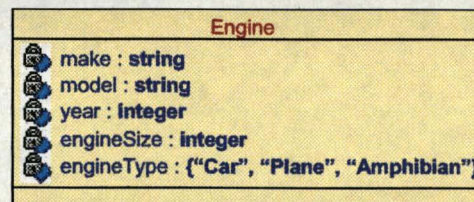


Figure 2.12: ENGINE class

Alternatively, it is possible to create as many classes as types of engines and declared each of them as subclasses of a more abstracted **ENGINE** class:

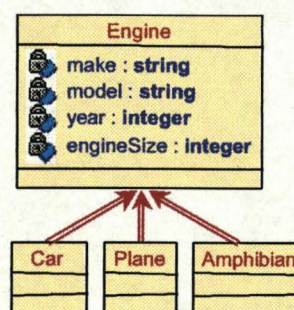


Figure 2.13: ENGINE hierarchy

The design choices arise when the classification can be made depending on many factors. Meyer stated that a common mistake is referred to as the *taxomania* mistake. A simple boolean or enumerated attribute such as a car's colour, is used as an inheritance criterion even though no significant feature variants depend on it.

- **Data-centered or functional-centered:** traditionally, designers were concerned with the data structures of entities, particularly for database schema design [Chen81]. With the introduction of object technology, the consequence was that resulting classes were used merely as a facility for encapsulating data structures with little behaviour attached, therefore giving no additional benefits from the traditional view. On the contrary, when the emphasis was functional-centred, the resulting classes were more used as a grouping unit facility and did not reflect a real-world object. Abstractions were not captured and objects were seen as a set of procedures.
- **Class size and class abstraction:** the size of a class should not be relevant when building a model. However, it can be used as a good indicator of excessive or non-effective class design. For example, if a class size, in terms of number of methods, is higher than the average number of methods, for the whole set of classes in the system, this might indicate a potential wrong decomposition of the class considered. In such cases, the class might do too much. On the contrary, when a class includes a small number of methods, it might indicate a strong dependency with other classes. Often, such classes require to be redesigned as they may capture many abstractions or none at all. Such problems relate to the notion of *class cohesion*. In the case of base classes, the application of generalisation is done in a bottom-up fashion and common properties should reside in classes situated in the top part of a hierarchy. If a class holds many abstractions or is not refined enough, it probably contains a subset of properties which would not be applicable to all its subclasses. A consequence of such a situation is that the cancelling of property inheritance, also called *disinheritance* is likely to happen in lower classes.
- **Inheritance or delegation:** inheritance is one possible mechanism to share information between objects. The *delegation* mechanism is another possible way to achieve the same although the underlying semantics is based on a client-server model as opposed to the inheritance model. Due to the similarity of the resulting consequence of both mechanisms, a common mistake is to use inheritance when delegation was appropriate and vice-versa. For a caller object, the delegation mechanism consists of requiring other object capabilities to realise a wanted task which will return the result back to the caller once completed.

- **Inherit or disinherit:** Firesmith [Fir95] recommends that no cancellation of inheritance of properties should be done in a class, also referred to as *uneffecting* properties in the Eiffel terminology. This feature is contradictory to the notion of inheritance. On one hand, inheritance proposes the heritage of properties and, on the other hand, it is possible to not inherit as well. Dealing with disinheritance constitutes an entire part of the design process and contradicts the natural mechanism of class hierarchy extension. During implementation, the detection of cancelled properties is not straightforward without any tool support.

Although arguable, the notion of inheritance, in object technology, has been considered as one of the major novelties introduced to software development. The notion of redefinition of properties has contributed to its inherent complexity and difficulty to control the property inheritance scheme. Instead of purely and simply inheriting existing behaviour from a parent class, the child class has the possibility of mutating the behaviour's internals in order to adapt it to its own purpose. The next section gives a presentation of redefinition where the main categories are highlighted and will serve as basis of study for the remaining part of the thesis.

## 2.2. On the notion of redefinition

*"Children have more needs of models than of critics"* – Carolyn Coats

### Why redefine if inherited?

Redefinition is the fundamental mechanism that provides the mutability and adaptability aspects of methods in class hierarchies. When the inheritance relationship is used between classes, the subclasses of a parent class can use, extend, replace or ignore the set of behavioural properties defined in all its parent classes. In the case of replacement of the behaviour, this is referred to as the *method redefinition mechanism*. Redefinition can generate many behavioural and conceptual inconsistencies in a class library. The mechanism is still controversial [Mey88, Rum96, Tai96] and there is a lack of understanding on the full effect of the mechanism on the overall class hierarchy.

### Use of redefinition

In the literature, method redefinition is generally described as a syntactic language feature [Boo94, Gra94, HenEdw94, Liu96, Mey88, Rum96, PapLeJ97] rather than a design concept. To date, the implications of use of method redefinition are unclear. This thesis addresses such problems in focusing on a conceptual description of redefinition and in providing the methodology and tools to analyse the behavioural aspect of class hierarchies.

In current OO methodologies, designers rely on lists of guidelines to validate the use of redefinition. In theory, designers should ensure that the semantics of a method remain the same if

changes are made to its implementation. Often, the examples of method redefinition relate to an illustration of the concept of polymorphism [Mey88].

An example list of Rumbaugh [Rum91]'s recommendations on redefinition is as follows:

- Query operations should not be redefined.
- A redefined operation should not restrict the semantics of the inherited operation.
- Redefining operations should never change the protocol or the underlying semantics of the inherited operation.
- Separation of interface from implementation should help in detecting useful redefinition.
- If all inherited methods are redefined, the subclass is wrongly subclassed.
- If no redefinition is used, it suggests that polymorphism is non-existent.

To date, designers can only rely on such guidelines, similar to the above-mentioned, for using the redefinition feature. Although a detailed description of the mechanism can be found in case study examples, there is a lack of methods for the validation of its use in class hierarchies when many levels of depth are present. Firesmith described a set of inheritance guidelines which gives practical advice concerning a class hierarchy design [Fir95]. However, in practice there are no guarantees that a given case of method redefinition is correct. A system can actually work without satisfying the guidelines or essence of inheritance. Design rules exist, but there are still various problems for which only designer's experiences and intuition help. In those cases, it is argued that assessment techniques come into the scene and are able to provide useful help in identifying and understanding the problem and suggesting design improvement directions.

This section analyses the different redefinition categories in the view of identifying the essential quality attributes to be considered within the measurement plan. Emphasis is given to the identification of possible uses of redefinition and the reasons why the mechanism may generate conceptual design problems. Also, it is essential to understand the consequences of use of redefinition in order to recognise potential caveats in complex structures such as class hierarchies.

### 2.2.1. The redefinition principle

*"Redefinition is an important semantic mechanism for providing the object-oriented brand of polymorphism"* – Bertrand Meyer [Mey88]

The basic principle of method redefinition is simple. In a class hierarchy, any class which has one or many parent classes inherits the properties of its nearest parent and, by transitivity of inheritance, the ones from further ancestors. In a multiple inheritance case, the parents are situated in different branches (see section 2.1.8). Method redefinition is a syntactic programming language



facility that preserves the original method name when the body changes. Conceptually, one of the main reasons for using redefinition is to provide the flexibility of defining a different implementation if needed, thus the ability for an original method to hold many forms in many subclasses of the same parent class. Such methods are called *polymorphic*. At run-time, the correct behaviour will then be dynamically bound to the object which receives the message (the receiver).

The principle of redefinition is also referred as *name overloading* or *overriding* as it exists in Algol 68 or Ada. Notice that the *renaming* mechanism provided by the Eiffel language is different from redefinition. The idea is simply to provide aliases to the same inherited feature. It is a syntactic mechanism which prevents name conflicts in a multiple inheritance situation.

The change of the semantics of the behaviour when using method redefinition is the fundamental issue. Meyer claimed that this situation is contrary to the spirit of redefinition and provides the concept of *assertions* to tackle the semantic problem. Constructions such as *preconditions* and *post-conditions* are effective ways to realise the specified contract and ensure that any subclasses inherit the correct behaviour.

The next section gives the necessary conditions that enable method redefinition to take place.

### 2.2.2. Conditions for realising method redefinition

In order to realise method redefinition, there must be an inheritance relationship defined between two or more classes. Suppose that a superclass *AParent* is defined as

$$AParent = \{|\emptyset|, \langle mthInParent \rangle\},$$

then a subclass would be defined as

$$AChild = \{|\emptyset|, \langle \langle mthInParent \rangle \rangle, \langle mthInChild \rangle\}$$

where *mthInParent* is inherited and *mthInChild* an additional feature of *AChild*.

From the formal definition of inheritance and the property ownership transfer given in sections 2.1.5 and 2.1.6, a method can be redefined only if it is first inherited.

Thus, for a class  $C = \{|\emptyset|, \langle \langle m \rangle \rangle, \langle \emptyset \rangle\}$ , *m* is inherited if and only if:

- *m* is defined in, at least one of its superclass(es).
- *m* is publicly accessible by the methods in *C*.

If a method *m* in a class *C* is redefined, it can be considered as a new property of the class as it physically extends or replaces the original method. In the case of methods originally declared as *abstract*, the subclass must provide the body of such methods, thus a completely new property for the subclass.

---

A method  $m$  of class  $C$  is redefined if and only if:

- $m$  is an inherited method (1),
  - $m(C)$  signature is the same as in its original definition (2)<sup>10</sup>,
  - $m(C)$  implementation is either, replaced, extended, or provided (3).
- 

If  $mthInParent$  is redefined in the class  $AChild$ , then the class becomes:

$$AChild = \{\emptyset, \langle mthInParent, mthInChild \rangle\}$$

The parameter list<sup>11</sup> and body of the methods may have changed. Therefore, the  $mthInParent$  method is considered as a new method for the class with the particularity of inheriting a portion or none of its parent's definition. Usually, redefined methods add specialisation to a class, thus enhance its behavioural aspect.

In a class hierarchy, it is expected that methods would be mostly reused or extended. By consequence, the leaf classes are potentially inheriting a large number of methods. This is graphically illustrated in the next section.

### 2.2.3. Descendants' heritage extent (hierarchy collapse)

Suppose that a branch of a hierarchy collapses. Instead of having many classes in the branch, an equivalent behavioural construction would be to regroup all the methods from all classes in the branch into a single larger class. This process is known as *flattening* [Hen96]. In the flat class, all methods are unique and for the ones redefined within the branch, only the latest version appears. In the Eiffel development environment [Mey&al95], there exists one such functionality that helps the designer to browse and understand the class's internals: the *flat form* view. Amongst other features, a class, in its flat form representation, displays the list of inherited properties from all its ancestor's classes within the same level. Therefore, a list of accessible features and their origin is made available in the flat form view, facilitating the search for suitable class properties. It should be noted that the flat form only displays the latest version of its properties, redefined or not. Therefore, all intermediate implementations are not shown. This method is sometimes convenient for assessing behavioural characteristics of the hierarchy.

In Figure 2.14 the extent of the expected descendant heritage is modelled for the  $Child$  class. When a class inherits properties from its parents, all of them are virtually present in the class plus the delta parts:  $x$  and  $y$ . In an *is\_a* relationship, part of the inherited properties is reused without modification and another part is redefined.

---

<sup>10</sup> In C++, name overloading permits a redefinition of the parameter list only.

<sup>11</sup> Note that, in Smalltalk, as the name also defines the parameter list, only the body is allowed to change in the case of a method redefinition.

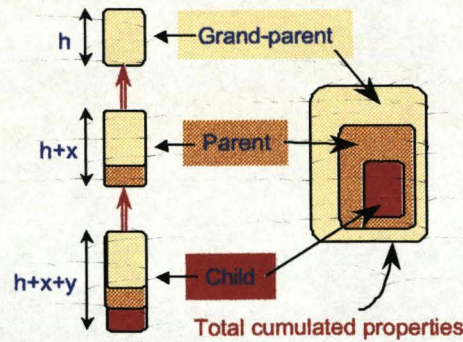


Figure 2.14: Expected descendant heritage extent

**Example of code for inheritance reuse and extension**

```

class C1 {
public:
    int add5(int n) { return (n += 5); }
}

class C2 extends C1 {
public:
    int exponentM(int n, int m) { return ( (super add5(n) )^m ); }
    int square(int n) { return ( (add5(n) )^2 ); }
    int cube(int n) { return ( (add5(n) )^3 ); }
    int add5(int n) {
        if (n < 0) n = 0;
        return ( super add5(n) );
    }
}

```

The `add5C1()` method is publicly inherited in the class `C2`, therefore reusable. The measure of amount of reuse in OO systems strictly depends on the definition attributed to the term “reuse”. Code reuse can be interpreted in different ways. One possible measure of reuse is to simply count the number of times an inherited method is referenced within each of the subclasses. In the above example, the `add5C1()` method is called twice in the class `C2`. The method calls are detected by the keyword `super` which means that the parent's method is called. However, the counting strategy does not specify whether indirect calls should be included or not. Indirect calls are made through intermediate methods such as the ones in the `square()` and `cube()` methods. Counting such calls would raise the number of calls to the inherited `add5()` method to four. Such situations demonstrate, for the reuse criterion, how ambiguous an empirical evaluation could be when its definition and semantics do not cover a particular case. Another important case is the fact that `add5C2()` redefines (in this case, extends) the inherited implementation. Therefore, it is arguable

that such a redefined method can be considered as a new method to the class C2, in which the first counting method remains valid.

The above code example and Figure 2.14 illustrated the use of redefinition in the case of extension; however, there exist other redefinition categories. This is detailed in the following sections. As the aim of this thesis is to assess the different uses of inheritance and its correctness, emphasis will be given to the redefinition categories that present potential problems from a design perspective.

#### 2.2.4. The main redefinition variants

Despite its very important role in a class hierarchy design process, the term redefinition is actually used in a confused way. Sometimes, it is referred to in the sense of method extension and other times in the sense of method replacement. Although, in both cases, the method is effectively redefined, their aims diverge completely. Method extension permits the reuse of the inherited property whereas method replacement stops the heritage of a parent property by not using it and replacing completely the inherited implementation with a new one. Method replacement seems intuitively unnatural unless as used in the case of a polymorphic method. For example, consider the following Smalltalk Collection branch:

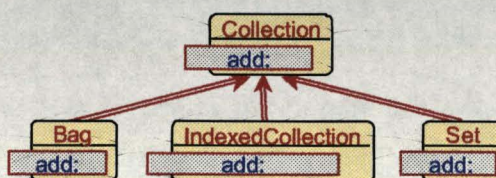


Figure 2.15: Part of the Smalltalk Collection branch

The `add:` method of the class `Collection` is declared as abstract, therefore it is necessary for the subclasses to provide the implementation of the class. In such a case, redefinition is correctly used.

In order to assess the "goodness" of a class hierarchy in terms of criteria such as coupling, cohesion, reuse or inheritance, it is important to understand and define what characteristics are to be measured. The hypothesis is that a high level of redefinition or its variants suggests a possible conceptual design problem in the hierarchy e.g. a class which was wrongly subclassed. The redefinition of a method will be assessed regarding its main variants [Lew95b] described in Figure 2.16.

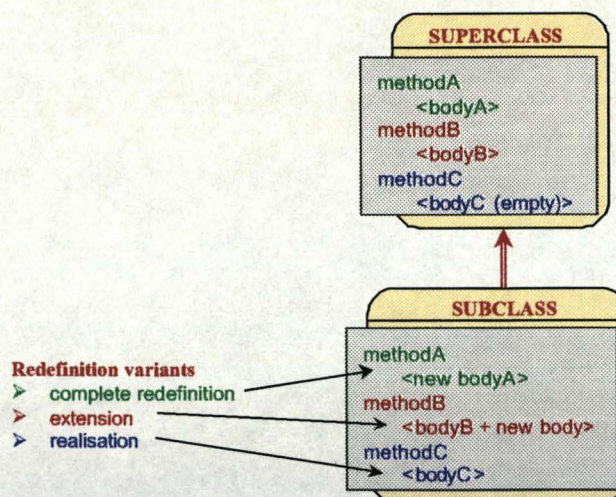


Figure 2.16: Different types of methods redefinition

The SUPERCLASS's methods are assumed to be publicly inherited. In SUBCLASS, the first case of the redefinition variants depicts an arguable case of inheritance where a complete redefinition of a method is done. Whereas the last two cases, extension and realisation, represent the recommended use of property inheritance. Cancellation of methods is an example of complete redefinition that restricts or stops the inheritance scheme. An extension to the implementation of methodB permits the reuse of inherited code and the addition of extra code which makes the subclass a specialised version. It should be noted that all cases of inheritance fall under one of the different types of method redefinition mentioned.

### 2.2.5. Remark on *super* method calls

This section highlights the fact that the type of calls to inherited methods may greatly affect the control of the behavioural inheritance.

When defining an *is\_a* relationship between two classes and providing that the parent class does not restrict the scope of inheritance, the subclass is offered the possibility to accept or refuse the parent's properties. In Smalltalk, a reference to the superclass properties is done by addition of the pseudo-variable [GolRob85] *super* in front of the property referenced. The default inherited feature called is always the one which was lastly defined or redefined in one of the superclasses. Thus, if many versions of the same feature exist in the descendants, the latest implementation is used. This will be referred to as the *direct* inherited property as opposed to other versions defined higher in the hierarchy. Note that the expressions *direct* or *immediate* classes will be used as opposed to *further* or *distant* classes. It is the *method-lookup mechanism* [GolRob90, Riv96] which allows the execution of the correct version at run-time.

In the C++ language, it is possible to call any publicly declared property of a superclass using the scope operator e.g. `classTest::methodA()`. Thus, if a method has been redefined many times in subclasses, any of the implementations can be recalled from the leaf class in specifying the above-

mentioned scope operator which is, in fact, the class name followed by the property name, separated by two semi-colon characters. These types of calls that clearly deviate from the expected inheritance scheme adds complexity to the understanding of the overall class hierarchy, thereby compromising its reusability or maintainability. The previous situation also illustrates a case where the complexity of the coupling between the parent and child classes is increased. This has been recognised as one of the major problems of inheritance hierarchies [Bri&a195, ChiKem91, ChiKem94, HarNit96, Hen96, Hit95, LorKid94, Mey88, Whi97]. The less coupling between classes, components or modules, the better. Further research is needed in this area and is outwith the scope of this thesis.

The next section describes one of the most debatable cases of inheritance which is referred to as *disinheritance*. This study is crucial for the understanding of the design characteristics that are involved in a measurement programme.

#### 2.2.6. Disinheritance and inheritance refusal

Two problematic cases of property inheritance arise when a parent class disinherits its child classes or when the child classes refuse the inherited properties from its parent classes. A conceptual approach is taken in this section in order to shed light on the reasoning behind such situations. It is argued that such cases of inheritance use are one of the main causes for complex inheritance hierarchies and are often related to fundamental design problems.

Inheritance aims at propagating ancestors' properties. If the properties are required to be known only by the class or by a subset of its heir classes, the access and visibility of the properties are controlled by the encapsulation mechanism. However, such inheritance situations can be disturbed by explicit or non-explicit restrictions as described below:

- **Parent classes impose restrictions for future child classes:** the Eiffel language provides a particular construct which allows, in a class, explicitly naming the heirs for which a set of its properties will be made available.

```
class EMPLOYEE
    export {MANAGER, DESIGNER} salaryGradeA end
end
```

In the above example, the `salaryGradeA` method will be accessible to only the `MANAGER` and `DESIGNER` subclasses of `EMPLOYEE`. The main benefits of such constructs bring rigour to the specification of a class. The property inheritance scheme is explicitly stated within the class. However, it also adds additional complexity for the management and control of behaviour in a class hierarchy. Exporting properties to only a subset of classes simply means that the concerned properties are not relevant or even not applicable to the other remaining subset, thus suggesting a design subclassing problem. One classification might satisfy one

criterion while violating another criterion, most of the times because of particularities which prevent obtaining a satisfying design. For example, the case of an ostrich being a bird or not (i.e. OSTRICH *is\_a* BIRD?) has been studied by many authors. The peculiarity of an ostrich not being able to fly but still being categorised as a bird in animal taxonomy raised the problem. Meyer proposed a solution using an inheritance construct whereby pre-conditions are applied to properties. To simplify, an ostrich would not satisfy the pre-conditions required for the fly method, thus the method would not be accessible to ostriches. By consequence, the evaluation of the goodness of inheritance use should also take into account those particularities when interpreting the values obtained from metrics. The assessment of redefinition is part of the design trade-off.

- **Child classes refuse a visible and accessible property of its parent class:** this can be achieved in two ways:
  - \* Ignoring inherited features: in this case, the features are simply not used i.e. not referenced in the class. Usually, in a class hierarchy, the leaf classes are the classes which encompasses all the knowledge given by the ancestor's classes. In this perspective, intermediate classes are just passing inherited properties to future subclasses and finally to the leaf classes. However, if an inherited property does not conform to an intermediate class e.g. a method which does not apply to instances of the class, the inheritance relationship might be questionable. Such situations do exist in current class libraries. This clearly illustrates the dilemma between the intrinsic genericity aspect of class libraries and the specificity aspect required to produce a solution to a design problem (see section 2.1.3).
  - \* Redefining the property: this category of redefinition is of particular interest for this work. If many cases of complete method redefinition exist in a subclass, it suggests a potential design problem whereby the subclass might not hold a correct inheritance relationship with the parent class, therefore a case of a class wrongly subclassed. Incremental development sometimes leads to inheritance complications and difficulties in controlling the extent of multiple changes of a method's implementation down a branch of the hierarchy. For example, it is common to add new methods at higher levels of the hierarchy, so that all the subclasses can benefit from the new method introduced. Assuming that the semantics of the method remain the same for all its descendant classes, different implementations might still be needed. In fact, the property redefinition happens because the parent class does not provide the desired behaviour, thereby requiring the replacement of the inherited implementation. It is precisely the difference of semantics between the parent and the replaced method's implementation that poses the fundamental design issue. The "Design by contract" methodology [Mey97, Ste&al96] aims at tackling such problems.

Clearly, there exist design solutions which fit the requirements but contradict inheritance. Therefore, this strongly suggests that inheritance is not always the most appropriate concept for solving certain business requirements.

While the above described redefinition models provide a flexible way to address particular design problems, they may also introduce inconsistencies in the design. The remaining part of this thesis investigates possible approaches to evaluate the correctness of a design regarding design inconsistencies that are introduced by unclear uses of the method redefinition mechanism. Given that a design solution may satisfy some of the design criteria while compromising others, it is fair to search for the best compromise, and admit that a design may not satisfy 100% of the criteria required during the assessment of the design.

This section introduced the main redefinition variants and their respective properties. They constitute strong candidate subjects for the assessment of the behavioural aspect in class hierarchy. Rating the presence of each category gives indications of the type of redefinition used as opposed to what is theoretically expected or recommended.

It can be argued that obscure uses of inheritance ought to be detected at design phases; however, this is not straightforward due to the inherent complex hierarchical structures that inheritance produces.

In the previous sections, the inheritance mechanism has been presented. In order to build a measurement plan to assess the correctness of inheritance uses, it is essential to recognise what constitute good, bad, expected or unexpected uses. *Heuristics* address such issues in recommending appropriate uses of object concepts and in helping the design decisions for trade-offs. Heuristics are investigated as a means to identify correct and incorrect uses of method redefinition and are aimed at providing suggestions where design improvement is possible. In section 3.4, it is also shown how the interpretation of metrics can be based on existing guidelines to address identified design problems.

### **2.3. Heuristics or guidelines for object-oriented design**

A consequence of the major hurdles [AksBer92] encountered during the design phase concerns the capture of the rules of OO design called *heuristics or guidelines* i.e. recommendations on the correct use of an aspect of object concept or mechanism. In general, heuristics describe the *what* without telling the *how* or *why*. Heuristics are orthogonal to a methodology in the sense that they exist as a repository of good advice to be used as a checklist. This repository usually comes from the extraction of all rules and constraints recommended in a methodology to form a summary synthesis.

Given the multiple inheritance models (section 2.1.3), an assessment of inheritance requires further precisions on the intention of the designer e.g. the inheritance model, the problem tackled and the



expectations. These expectations may originate either from the OO methodology or announced by the designers. In our case, if it were to assess the method redefinition mechanism, one would state not only the goals of the assessment but also what is considered as good or bad. To do so, the heuristics constitute a possible approach for the designer to state the hypotheses, assumptions or general recommendations regarding the subject assessed. Reference to such heuristics is valuable as a design aid tool; however, it requires to be supported by a quantitative process that permits the validation or invalidation on the correctness of the design.

This section gives a general overview of heuristics. It is shown how the technique can be used as a design technique, thereby providing an opportunity for defining the intended uses of inheritance. The benefits, applicability and restrictions of heuristics are outlined.

### 2.3.1. Definition and purpose

*Guidelines* for OO design are, by definition, aimed at guiding the process of design. Sometimes, they are referred to as *principles* although this term implies strict respect for the topic described. A basic definition of heuristic is as follows:

---

**Heuristic** [Fol97]:

A rule of thumb, simplification or educated guess that reduces or limits the search for solutions in domains that are difficult and poorly understood. Unlike algorithms, heuristics do not guarantee optimal, or even feasible, solutions and are often used with no theoretical guarantee.

---

From a software engineering viewpoint, it is surprising why the interest for guidelines has increased from the birth of the OO paradigm. One could question if there is a need for those design guidelines as all details should be already explained and examined in the OO methods. A first answer can be found in observations made from past experiences. As many factors may influence the profile of an OO model, it is the designer's responsibility to ensure the best possible compromise for a good OO model. Each design solution corresponds to a particular design problem space. This is the reason why designers ought to capture the commonalities between each design context, so it can be reproducible. Such difficulties are reflected in the design decision making process. Thus, heuristics originate from the intention of designers to describe good uses of the OO concepts. For instance, the use of abstraction or generalisation varies according to the designer. When many approaches exist to solve a design problem, designers can rely on heuristics to guide their decisions. Riel [Rie96] described his work as an attempt to capture this subconscious list of heuristics which guru designers use to "validate" their design. If the heuristics pass, then the design feels right, and vice-versa. Note that in any cases, humans' mistakes still represent one of main sources of errors. Heuristics may also state the conditions under which the application of a technique or a mechanism will exhibit good quality factors. In general, heuristics are considered as part of OO design methods although they may not be specifically referred to as heuristics.

The development of large software systems provided experience in producing lists of guidelines for good design. Even though they remain textual explanations, their application whilst building models help increase the level of quality of applications with respect to reusability and maintainability [Fir95]. Heuristics for OO design are categorised according to the various aspects found in OO, and often address unclear or imprecise design features i.e. use of the inheritance relationship for subclassing or subtyping. Recently, Riel [Rie96] proposed more than sixty heuristics which cover most aspects of OO design from objects, classes, the different kind of relationship to a complete OO model. The author even mentioned that the heuristics provided are to be only considered as rules of thumb and not as rules which must be followed. Those heuristics exist for the sole purpose of warning when the design does not satisfy a given one. However, the decision will always be up to the designer for further actions if judged necessary.

The main characteristics of design heuristics are outlined below:

- Non-formal.
- Language dependent or independent.
- Rely on observations from past experiences.
- Outline the main idea of a concept.
- Give an interpretation on the proper use of a technique or mechanism.
- Non-compulsory.

In general, heuristics are recognised as good indicators of anomalies or infringement of design principles. For example, a class hierarchy that is extended in width rather than in depth illustrates that the inheritance mechanism is used in only one particular aspect and that redundancy of services might appear in the subclasses. Ultimately, design guidelines provide directions to tackle design problems.

Examples of heuristics' classification from different authors can be found in the Appendix.

Heuristics may be used in a wide range of topics from conceptual design to programming language constructs. However, one particular limiting aspect of heuristics is that they may be subject to various interpretations. In such a situation, their application may also be compromised. The next section relates such issues.

### 2.3.2. Interpretation

On one hand, heuristics' informal description underlines the fact that they should be manipulated as good design advice rather than strict rules. On the other hand, the definition also specifies that they may be open to many interpretations. In general, heuristics recognise the good or bad practices in design but do not suggest approaches to reach that aim. Heuristics that encompass a

subjective characteristic are particularly questionable. For example, Firesmith's [Fir95] guideline G-30 states: "*Avoid inheritance structures that are too shallow or too deep*". It argues that inheritance hierarchies are considered shallow when they are less than three levels deep and deep when they more than seven levels deep. Those assumptions are indeed debatable and highly dependent on the domain and the designer's experience. On the contrary, Kennedy [Ken92] promoted a deep hierarchy approach based on abstract data types. By following his guidelines, a designer would not face the important problem of providing too much or too little information within a class. A deep hierarchy is effectively breaking up the problem into many classes. Another variation of the same principle for inheritance is given by Riel [Rie96]: "5.4: *In theory, inheritance hierarchies should be deep*" and "5.5: *In practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory*". The application of heuristics still remains difficult because of their open interpretation.

Although valid, heuristics may not be relevant in all design situations as it depends on many factors such as the requirements and criteria of the application. For instance, consider the following contradictory guidelines:

- *Class coupling is not recommended because it creates a dependency link between the classes.*
- *Commonality in data, behaviour and/or interface should be factored out to the higher levels of the hierarchy.*

The second guideline encourages the creation of abstract classes in higher levels of the hierarchy, therefore is in favour of decomposing and organising the behaviour in appropriate abstract classes. Creating many levels of abstraction implies an increase in the number of classes in the system. So, when instances of a class are created, they rely on other information from other classes, therefore a possible increase of class coupling as well, which is contradictory with the first guideline.

Another difficulty in using heuristics is that exhaustive lists of recommendations seem to be adopted sparingly in companies and therefore, are under the influence of the practices in that environment. Frequently, recommendations are made for OO programming languages in order to generate some sort of uniform programming culture which makes easy communication between developers.

Riel [Rie96] argues that the designer does not get a prioritised ordering of the heuristics. Instead, the sense of priority comes from a combination of the application domain and the user's needs. Therefore, this suggests that the representation of heuristics should be either problem-based or characteristics-based, thus encouraging classification. The application of heuristics or guidelines is mainly requirements and constraints driven.

It is clear that heuristics may not be as beneficial as expected for the reasons that there are no supporting techniques or tools to verify if the heuristics are realised. To avoid the above-mentioned problem of heuristics' interpretation in this thesis, attention will be given to heuristics

that address specific design issues rather than general ones. In such a case, it is believed that heuristics will permit a fairly accurate description of the problems of inheritance, thereby facilitating the use of quantitative measures on the design attributes. However, the use of quantitative measures will not remove the subjectivity aspect of the heuristics, but rather only provides the basis for development of non-subjective assessment.

The next section illustrates an example use of heuristics.

### 2.3.3. Example of heuristic's application

#### Class correctness

Different design solutions exist for the same problem. For example, Rumbaugh proposed that a single class with appropriate attributes e.g. instance variable of basic type or of aggregate type should be considered when the potential subclasses do not hold different forms [Rum93].

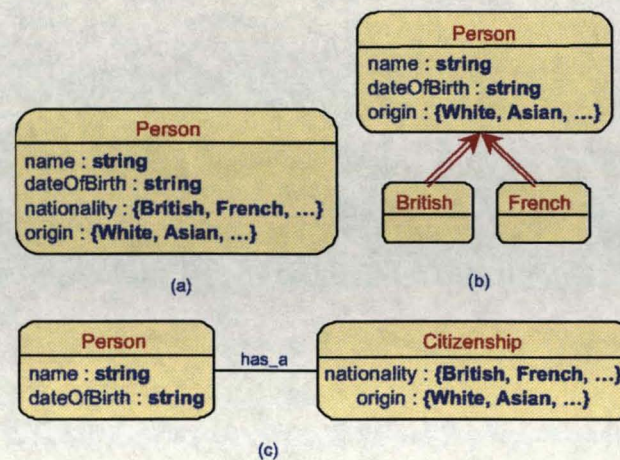


Figure 2.17: Three possible designs for the class Person

Figure 2.17 shows three different models representing the same information about a person. Applying Rumbaugh's above-mentioned guideline, the design (a) is preferred because the creation of two subclasses `BritishPerson` and `FrenchPerson` do not add further information to the design as in design (b). In addition, the same problem occurs for representing the origin of a person. In fact, a much more flexible design is shown in (c) where the information about any kind of citizenship is modelled as a `Citizenship` class and any person holds a link to this information. Suppose that depending on the nationality of a person, there exists a different set of regulations. A possible solution to keep track of the regulations would be to store them as behaviour in the class `Citizenship` (design (c)). The following guidelines are satisfied as well: "keep related data and behaviour in one place" and "descriptive attributes should be modelled as properties" [Rie96]. The appropriateness of the `Citizenship` class (as opposed to an attribute) was justified by the presence of behaviour for the different nationalities represented. `Citizenship` class can therefore be used independently in other contexts, resulting in a de-coupling of information among classes.

Note that, with the help of the heuristics, a model can be successively refined in order to solve the same problem in improved ways. The first important step before applying heuristics to a design is to select the relevant ones for the project. Then, a priority order can be attributed to each identified heuristic within each of the categories.

### Modelling gap: translation from textual analysis to design to implementation

It has been generally recognised that in the early phases of the software development life cycle the transition from the user requirements to the specification phases raises the problem of capture and comprehension of the users concepts. This has been referred to as the *mapping* and the *modelling gap* problems (Figure 2.18).

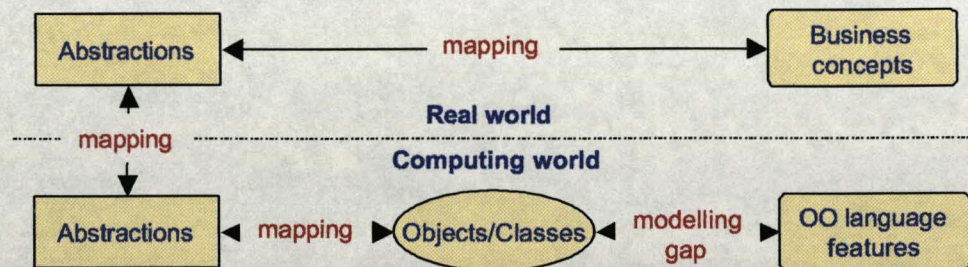


Figure 2.18: Mapping and modelling gap

This example illustrates a well-known application of heuristics or guidelines in order to find relevant objects from a textual analysis task. The early work on the identification of objects in a system is due to Abbott in 1983 [Abb83]. His idea was to extract the objects and methods from the textual specification of the problem based on simple rules or guidelines. With a direct mapping of the grammatical type of words to OO concepts it is possible to obtain a first object model.

Part of speech	Model component	Example
Proper noun	Instance	John
Improper noun	Class	company
Doing verb	Operation	lead
Being verb	Classification	is a
Having verb	Composition	has a
Stative verb	Invariance-condition	have bonus
Modal verb	Data semantics, precondition, post-condition, or invariance-condition	retires at 65
Adjective	Attribute value or class	
Adjectival phrase	Association, operation	
Transitive verb	Operation	
Intransitive verb	Exception or event	is able to

Table 2.1: Identification of objects from textual specifications

**Example:** Suppose that we want to model a *company* which employs a certain number of *employees*. A *manager* is able to lead many employees but an employee is responsible to a single manager. An employee receives a *bonus* on his *work anniversary*. In this company, an employee has the following status: junior, senior, project leader, manager and retires at 65.

In the above textual specification, the possible objects are shown in *Italic* while the relationships are underlined. A possible resulting object model would then be:

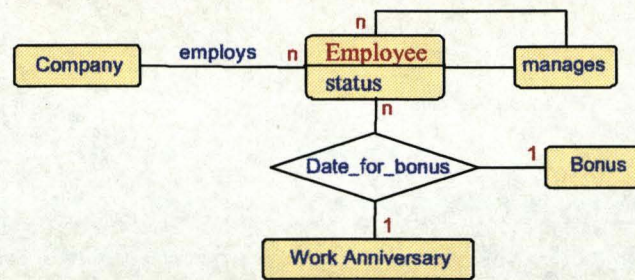


Figure 2.19: A company information system<sup>12</sup>

This section showed that heuristics constitute a useful informal technique to tackle design problems. Intuitively, it can be claimed that the human's common sense is the first form of heuristics. Heuristics give indications on the correctness of a design and can be used as a complementary technique to the design process. Therefore, it is a good candidate technique to investigate potential misuses of the redefinition mechanism. However, it has been previously stated (section 2.3.1) that heuristics do not guarantee solutions. The next chapter describes how the recent subject of OO design assessment can shed light on many design areas where suspected problems occur. It is believed that measurement techniques can support heuristics in the sense that it provides quantitative elements to identify the realisation of a heuristic. Thus, assessment techniques are envisaged as an approach to the validation or invalidation of the heuristics.

The following section focuses on measurement techniques in a general manner and describes the current state of research for the assessment of object oriented concepts. In particular, the process of measurement is detailed with the aim of identifying the different aspects for applying metrics to an object model. In this thesis, the use of metrics is considered in order to detect design defects using inheritance and suggest solutions to identified problems.

<sup>12</sup> Note that some assumptions were made before drawing the object model in Figure 2.19:

- A manager is an employee. The factorisation of features encourages genericity. Note that, if the manager attributes are to be represented e.g. salary, benefit and responsibility, an appropriate class would be required.
- The different status can be modelled using an attribute.
- Further generalisation of the model is not required but possible i.e. an abstract *Person* class could be introduced as the *Employee*'s superclass.

## 2.4. Assessment techniques

*"We must know what we are measuring"* – Norman E. Fenton [Fen91]

*"You cannot control what you cannot measure"* - Tom de Marco [DeM86]

Generally, assessment techniques are understood as the evaluation of the *quality* of a characteristic/attribute of an entity. Measurement techniques constitute the act of applying metrics to obtain measures (numerical value). Past experiences from the engineering discipline suggest that the science of measurement plays an important role in software engineering. However, software metrics have suffered from a lack of rigour which did not encourage its development and use until recently. A definition of measurement is as follows [Fen91]:

---

**Definition:** *Measurement* is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules.

---

An intuitive and informal design assessment technique relies on the designer's experiences and knowledge. Naturally, designers test and validate their designs against the requirements. However, as a design rapidly grows in size in terms of the amount of features such as class, attributes, methods, rules, constraints, etc, measurement techniques permit a deeper evaluation of an existing OO model.

The increased interest in metrics for OO has been significant for the last five years following the pioneering work of Chidamder and Kemerer [ChiKem91] with their OO metrics suite. Whitty's analysis of OO metrics literature [Whi96] not only showed that publications in this area have increased by a factor of nearly 10 from 1990 to 1995 but also that 45% of them concern product metrics applied to designs or code. Since OO programming languages encompass ready-made class hierarchies in their packages, there are opportunities for assessing both external and internal quality factors of class hierarchies, therefore a better understanding of the meaning and usage of the inheritance mechanism.

Assessment techniques help managers and designers to evaluate the quality of their projects [RosHya96] providing that the goals for measurement have been identified and described. Evaluation can occur at all stages of the development; however, for prediction, measures should be taken as early as possible in the process. Assessment can also be applied on an implemented application, therefore falls under the case of a re-engineering or refinement strategy of a current existing product. In such a case, it is interesting to know what areas need to be re-visited, taking into account any new requirements. Assessment techniques are divided into three categories of measures:

- **Processes:** software related activities which normally have a time factor such as specification, analysis and design,
- **Products:** deliverables such as documents, applications or other artefacts,

- **Resources:** any inputs to software production such as personnel, materials, tools and methods.

Although a measurement programme should bring benefits to the matter investigated, it does affect cost and schedule of the project. Cost issues are outwith the scope of this thesis; however, attention will be given to planning a metrics programme to be run within a project. As the design phase aims at producing deliverables in particular an object/class model, most of the rest of this document will put the emphasis on the product metrics category. Relevant metrics are the ones affecting the design phase.

In this thesis, the use of measurement techniques is envisaged as a means to assess the goodness of a class hierarchy with respect to the design criteria and design heuristics. This section explains the purpose of a measurement process, the expectations and benefits from the use of metrics and how a measurement plan is created.

#### 2.4.1. Roles of technical measurement

Fenton [Fen91] claimed that measurement has the two roles of prediction and assessment. The area of prediction relates to project management and comparisons are often made to previous project experiences. Fenton considered that prediction should remain the ultimate goal of measurement. Whitmire [Whi97] added another three roles to measurement and described the following:

1. **Estimation:** in many software projects, is it essential to identify previous experiences (from historical and environmental data of existing products) which can help in resolving the current requirements of the current project. The aim of estimation is to evaluate the resource requirements for future products.
2. **Prediction:** as opposed to estimation, prediction looks at values of product measures in considering values from existing products. Prediction is not so much based on historical and environmental data.
3. **Assessment:** from an evaluation perspective, the assessment process aims to compare values obtained from a product to previously defined values arbitrarily or not chosen as standards, benchmarks, projects goals, targets or customer requirements.
4. **Comparison:** the main purpose of comparison is to help in making design decisions i.e. trade-offs. Although assessment ought to compare values as well, comparison only takes into account measures taken from the product and not from predetermined values.
5. **Investigation:** in order to support or dismiss a hypothesis, measurement techniques can be used as a way of investigating unknown attributes or behaviour.

The assessment of software applications is expected to shed light on various quality criteria of a system. If the prediction of costs is possible, the budget planning process becomes easier and realistic [VerCor95]. Often, the assessment of the quality factors relies on measures taken from



internal factors. For example, assessing the overall reusability of code of a system, the reusability aspect must be assessed for all sub-levels in the architecture. Further details can be found in [DeM96, Fen91, HenEdw94, VerCor95], however this topic is outside the scope of the thesis.

The work in this thesis mainly concerns the assessment, comparison and investigation categories. A presentation of a software quality model is given in the next section to explain the essential process of creating a measurement plan.

#### 2.4.2. Software quality model

The success of a development and implementation of a metrics programme is based upon the underlying software quality model used to define the metrics themselves. In the same manner as for the software development phases, assessment methodologies exist and propose a step approach model from definition to implementation of a metrics programme. The well-established *Goal/Question/Metric* (GQM) [Bas&al94] model is such a model (Table 2.2).

Level	Assessment level	Description
Conceptual	Goal	Objects of measurement
Operational	Question	Characterisation of the way the assessment/achievement of a specific goal
Quantitative	Metric	Evaluation of the object to be assessed

Table 2.2: GQM levels

The GQM model describes a framework for developing a metrics programme. It provides a means of identifying and defining a concise plan detailing all necessary actions to identify, define and apply metrics, analyse and interpret the results and finally, return feedback to the designer. Figure 2.20 shows the GQM/MEDEA (MEtric DEfinition Approach) [Bri&al94] which is based on the GQM model. In this model, the steps are detailed and take into account possible external interactions or events which might affect the metrics programme.

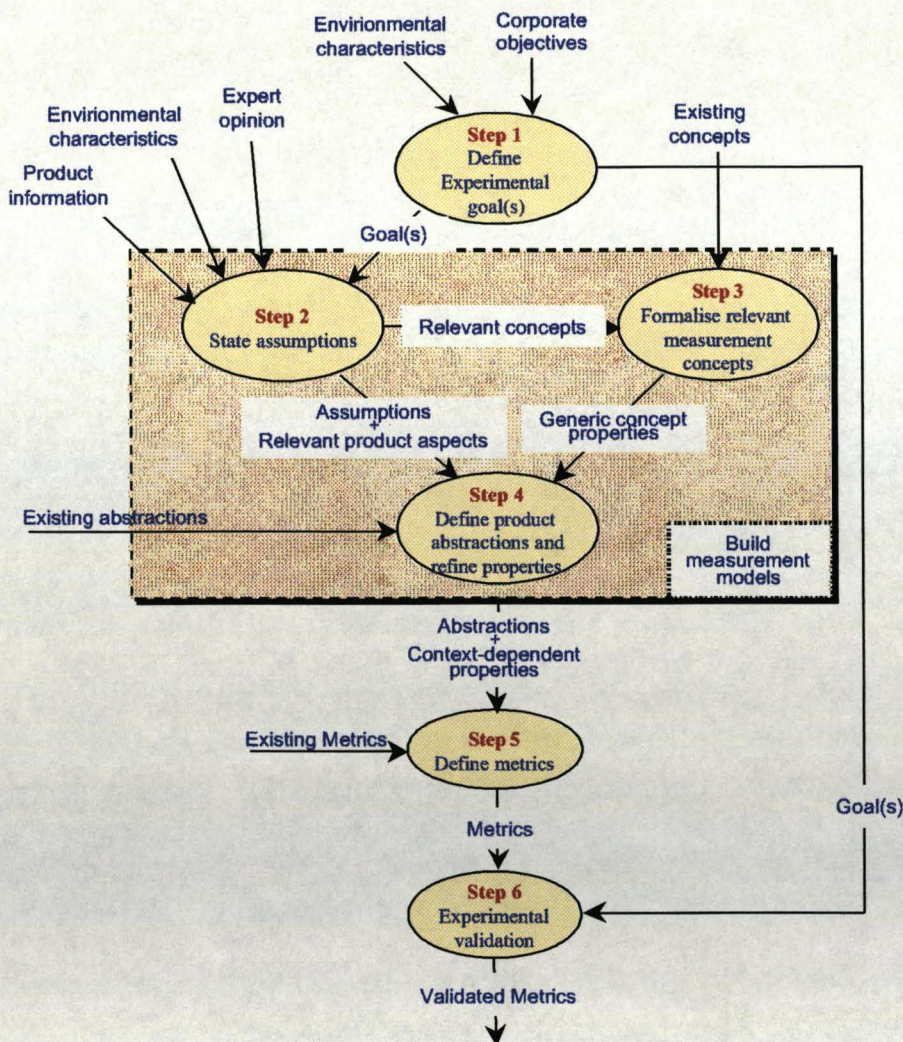


Figure 2.20: The GQM/MEDEA model

In many past experiments using metrics, the pragmatic approach raises the problem of validity of the results obtained by metrics derivation. A consequence of invalidated results is that wrong interpretation follows and finally unexpected conclusions arise. Validity of metrics is the first important concern addressed in a software quality model. Thus the danger of metrics is that they may not produce expected results on the characteristics measured i.e. wrong metrics. A possible definition of validity of measure is given:

---

**Definition** [Bak&al90]:

*Validation* of a software measure is the process of ensuring that the measure is a proper numerical characterisation of the claimed attribute

---

For a metric to be valid, it is generally accepted that the metric should embody a certain number of properties. The next section concentrates on properties that relate to the OO concepts.

### 2.4.3. Properties of software measures

Software metrics has suffered from criticisms concerning their real added value in managing and controlling software development. Nonetheless, when Basili [Bas&a194] proposed the GQM approach, he stated that metrics, in order to be effective, must be:

- Focused on specific goals.
- Applied to all life-cycle products, processes and resources.
- Interpreted based on characterisation and understanding of the organisational context, environment and goals.

It is interesting to note how well these three points summarise the expected properties of metrics. In the literature where criticisms have been made on the relevance of metrics for software development, one or more of these points are either omitted or unclear thereby casting doubt on the validity of metrics. For example, Hitz and Montazeri [HitMon95] categorised metrics depending on their causal effect on the design process. They argued that attributes can be divided into three kinds: “*fundamental*”, “*auxiliary*” and “*useless*”. In short, they stated that attribute selection often does not consider the first point of Basili’s metrics effectiveness criteria. Therefore, a metric measuring a wrong attribute does not invalidate the correctness of the metric itself. The SIZE1 and SIZE2 metrics proposed by Li and Henri [Li&a195] were challenged for their effective evaluation of costs per class as expected. If the metrics were to be minimised, the classes would be smaller. Providing that requirements remain the same, the number of classes would rise to fulfil them, therefore, generating an increase in the overall system complexity which in turn may increase overall maintenance costs. A metric is causal when a change applied to the attribute considered generates a different metric result. Therefore, it is expected for a metric to have the causality property.

In general, it is highly desirable for metrics to be:

- **Intuitive** (reasonable): when considering the assessment of an aspect A of an object model, finding related attributes or other aspects which are directly or indirectly related to aspect A should be intuitive.
- **Applicable or derivable**: the metric used must be applicable otherwise it is useless.
- **Related to the characteristic measured**: a measure of both the structure of the data and process must be included.
- **Independent of language**: a metric should capture a particular aspect of a concept or a concept itself, therefore should not depend on its underlying implementation.
- **Contained**: once defined a metric should be valid in the defined context but not dependent on conditions for its existence.

- **Basic or composite nature:** a metric is either basic or composed with other dependent metric(s).
- **Measures must be consistent:** if  $r$  is the result of metric  $m$  on an entity  $e$ , then if  $e$  changes,  $r$  should also have changed.
- **Represented at least on an ordinal scale of weak order:** the metric should be represented on a quantitative scale and not based on a subjective scale.
- **Additive** i.e. for two independent structures, the total complexity metrics should be the sum of complexity of the two individual ones.
- **Automatically collected by tools:** data collection is a time-consuming and expensive activity, therefore it is unrealistic to attempt any measurement programme if no tools are available to facilitate the process.

OO design methods do not include assessment techniques as part of the methodology. Instead, assessment methods are considered as additional techniques. The assessment for “goodness” of a design should be done under different perspectives in order to obtain valuable information for trade-offs. Thus, a possible definition of a good design is *“providing a set of design requirement criteria and associated priorities, a good design should mainly satisfy the few important ones without discarding the others”*. Unfortunately, current methodologies give a recipe for software design but there will always be a number of unpredictable error cases. In consequence, there is a need for a systematic design review process during or after the building of a model. Current design review methods include testing techniques and assessment techniques. Both these techniques help in detecting suspect designs once the problems are identified. Open interpretation of a concept leads to many design choices. To date, it is essentially a great effort of careful programming which avoids future maintenance costs.

The next section highlights the intrinsic internal quality factors of an OO design.

#### 2.4.4. Internal quality factors of OO design

*“Quality is relative to the intended use of the system”* – [Bar&a197]

Whilst researchers have focused on various software quality model that enable the construction of a measurement plan, it is equally important to review the aspects of an OO design that can be assessed. Given the software quality model described in section 2.4.2, recognising a good design necessitates first giving a definition of the qualifier: good. In a first attempt to assess a design, the designer’s intuition plays an important role. Often, knowing that a design ‘feels’ good or bad might be easy; however, giving an explanation of the grounds the conclusion was based on is rather difficult. Typical expressions include:

- *“It is good because the classes are reusable?”*

- *“It is good because polymorphism is used and common properties have been abstracted?”*
- *“It could be better because classes are too large.”*
- *“Should this information be represented as an attribute or a separate class?”*

In all cases, the conclusions remain vague and open to different interpretations. Often, this situation is due to an uncertainty of the attributes to be measured. According to Fenton, the clear distinction between a) product/process/resource attributes and measures, and b) internal and external attributes and measures is crucial before the identification of any possible candidate metric. For example, stating that classes are reusable requires further information on the kind of the reusability referred to. Is a class reusable because it has subclasses? This is not necessarily true in the case of implementation inheritance. In order to evaluate characteristics of an OO design, a detailed study of the object model and its context is necessary. Current assessment methods are based on measurement techniques applied to intrinsic characteristics of OO concepts. Assessing design characteristics requires the knowledge of the characteristics themselves with regard to the criteria to achieve. One possible approach is to use existing classifications of OO concepts in order to address a particular aspect of the system e.g. the quality factors.

Goodness of internal quality factors relates to the aspects being assessed. First, the OO aspects envisaged concern the stated criteria in the requirements. A life-critical application would be assessed for potential failure of the system. Second, concepts such as coupling, cohesion, reuse, depth of inheritance, hierarchy structure [ChiKem94, Bri&al94, LorKid94, Teg&al95] can be assessed to detect potential misuses. From a user viewpoint, software is considered good if it satisfies all the requirements. Internal quality factors concern the architectural, structural and behavioural design of the software. From a designer viewpoint, an example for which software is considered well designed is that the introduction of new parts in the system does not disturb the existing parts. Few papers have described concepts that have been wrongly used and for which metrics permitted assessment techniques to take place [BarSwi93, Bri&al95, LiHen93]. In general, obscure uses of OO mechanisms relate to either the structural or behavioural organisation of the classes in the model. Indeed, the architectural issues affect the overall quality criteria of the design. Thus, the motivation behind the assessment of OO models at various levels of complexity including system, class hierarchy or class levels.

In addition, metrics have also been defined for the internals of a class i.e. the instance variables and methods. Often, in a measurement programme a set of metrics is utilised for various reasons. When the metrics address related aspects of the design e.g. cohesion and message passing flow, complex dependencies between the classes may be explained. Tegarden et al [Teg&al95] proposed that the characteristics of a good OO design are identified by means of coupling and cohesion. They state that metrics can be categorised into two types of coupling: interaction and inheritance and three types of cohesion (service, class, and generalisation-specialisation). However, they identified four

possible levels of complexity which are the variables, the methods, the objects and the overall system.

While the "goodness" of an OO design can be measured by assessing its internal quality factors, a major component depends on the understanding and application of concepts provided in the OO paradigm. Determining a good set of metrics is strongly dependent on the interpretation of the concept measured. Fenton [Fen91] mentioned that measuring is not enough, one important aspect in an assessment process is also to state clearly the objectives, goals or specific motivations for establishing such a measurement programme. If software reuse is to be achieved it is essential that the structure and behaviour of the class are well designed. One way to tell about the "goodness" of a design is in recognising its "badness".

So far, the reasons and the process of building a measurement programme have been described. However, other considerations should be taken into account for the deployment of the programme. In particular, the next section highlights the dilemma between the desire of measuring at early stages of the design and the data availability issue. The practical issues in the application of metrics are explained.

#### 2.4.5. Data availability and metrics collection

Once the measurement programme has been identified and defined for the project, the application of the programme will start with the data collection phase. Data collection is recognised to be one of the main problems which can affect the success of the programme. If a metric ought to assess a particular aspect of the design, then the identification of the necessary attributes/properties related to the assessed subject should be available. Metrics claim to be implementation-independent (see section 2.4.3), therefore it implies that the code is not necessary for calculating the metrics. Indeed, an early assessment of the design, meaning that the information is available, favours early detection of potential problems. This is not always possible. Due to the incremental development process, any attributes are expected to evolve during design; thus assessing an unstable element is not good practice.

Without an automatic metric collection tool, it is unrealistic to perform a measurement programme. Deriving measures on an object model is purely a counting process. Classes, properties, data structures, meta-information and so on are parsed and required metric information is collected, then computed if necessary, and finally stored for later analysis. Not only is an appropriate measurement methodology necessary, but also tools [Bri96, BriCuc98, Fen91, LewSim98] are vital for a successful completion of a measurement activity.

To date, most metrication tools rely on source code for extracting measures. It has been criticised that taking measures when the implementation is done appears too late in the software development process. This is a valid criticism. Nevertheless, collecting metrics on source code still

gives much insight into both the design and most importantly, the language features used to implement a design solution. Often, there are no other choices. Therefore, an assessment of source code for design features should be considered as a valuable process for detection, investigation or evaluation purposes.

Current research has focused on the provision of generic tools which would be able to define, apply and analyse a range of measures in combination. This is still an active research area where more empirical studies are required in order to classify the different possible measures i.e. taxonomy of measures. So far, dedicated tools exist for a set of measures, often corresponding to an author's suite of metrics. Another area of research concerns the application of the metrics across languages. Languages have different constructs to implement the same concept, therefore different metrics are needed to cope with the equivalent syntaxes. Sometimes, such mappings are not straightforward or even possible. For example, metrics assessing multiple inheritance cannot be applied to single-inheritance languages such as the Smalltalk language.

The integration of assessment tools within CASE tools seems to be the natural solution to provide designers with complementary functionalities to assess a design while being built. To date, only few research projects have built specialised metrics tools for assessing internal quality factors of a design [BriCuc98, LewSim98]. Besides the metrics tool availability problem, the assessment methodology is still subject to debate. Measurement techniques are, without doubt, beneficial to designers and implementors but more empirical experiments are required to validate and quantify the quality of the measurement experiments themselves. In [Bri96], the main goals of automatic data collection tools are identified as:

- Simplification of data collection.
- Minimising the impact on the development schedule.
- Maintaining confidentiality of data.
- Providing value to target audience.

In this research work, the development of a metric collector tool is envisaged to support and demonstrate the use of metrics derived from an object model. The automation of the metrics collection process is crucial to the success of the programme.

In as much as the definition of the metrics is important, the analysis and interpretation of the metrics results is equally important for the extraction of meaningful feedback and possible actions for improvement. The issue of metrics interpretation is covered in the next section.

#### 2.4.6. Metrics interpretation

The application of metrics to an OO design aims at providing explanations or directions to the problem assessed. For instance, the discovery of unseen design problems may confirm the stated

hypotheses. The analysis and interpretation of metric results is problematic and sometimes unclear. Depending on the subject assessed and the purpose of measurement, the metrics results may not always guide the designers to the satisfactory conclusions. It is believed that such situations are due to various factors which can be decomposed into the following categories:

- **Metrics' definition:** the metric definition itself can be the cause of difficulty of interpretation, particularly when it does not measure the desired characteristic [Fen90, Hen96]. For instance, the LOC (Line Of Code) metric has been a subject of debate for its use in OO programming languages [Fen90, Hen96]. However, it has been generally recognised that the metric was not appropriate to the object model.
- **Identification of the purpose of the metrics:** although a metric may be completely valid, it may not be very useful. Collecting measures is part of the goals of a measurement programme, suggesting directions and solutions are the main outcome researched. For instance, there have been many attempts to provide measures on a particular aspect for the resultant software system. Often, those aspects are high level quality factors such as in the equation below[Hen96]:

---


$$\text{Quality} = \text{reliability} + \text{availability} + \text{maintainability} + \text{usability}$$

$$\text{Where maintainability} = \text{understandability} + \text{modifiability} + \text{testability}$$


---

It is argued that, metrics assessing an entire system are mostly beneficial if finer-grained metrics are jointly used in order to suggest more precise indications on where design goodness or badness occurs. In [Ban97], the proposed hierarchical object-oriented design quality framework relies on the decomposition and relations between high-level quality attributes and details of the structural and functional design properties.

The goals' definition is the first step of the measurement process [Bri&a194]. Assumptions about the characteristic measured are also defined. However, if incorrect assumptions are made, the interpretation of the metric results is also affected. Usually, assumptions relate to the interpretation of OO concepts, and therefore depend on the designer's experience.

- **Metrics' derivation:** often, because of an unclear description of the metric and its use, the interpretation of each can be wrong [ChuShe95, HitMon96]. In such a case, the user of the metric may elaborate many incorrect assumptions when ambiguity arises, thereby affecting the analysis of the results.
- **Metrics' results interpretation:** often relying on statistical methods, this does not seem entirely satisfactory [HarNit96] as the conclusions relate more to a mathematical model than to a design characteristic. On the other hand, averages or thresholds appear to be useful although based on an arbitrary choice for the value. The problem of interpretation is that without a reference or comparison value, the designer is left with an intuitive interpretation. For example, Henderson-Sellers [Hen96] stated that a first and simple approach is to infer relationship order



between the values e.g. a system containing 1000 classes is bigger than a system with only 20 classes. Then, the standard deviation of a particular measure from a mean value gives an indication on how different the measure is compared to an even distribution within a system. However, it is argued that such an interpretation is not appropriate in some cases. For example, the fact that a system has 20 methods on average per class would suggest that all classes should encompass around the same number of methods, otherwise it is considered as suspect. Note that this example assumes that the classes assessed belong to the same categories. In general, the inclusion of classes from different categories such as UI classes, facility classes, control classes, etc in the metric calculation raises the issue of interpretation of the results due to the fundamental nature of each.

To date, proposed software quality models only cover the first two points above described. However, it is the interpretation of the metrics results phase that provides the final conclusions, therefore it is vital for the success of the measurement programme.

Computing an average or a threshold constitutes another research problem for the metric interpretation. Generally, it involves the derivation of the metric on the entire system in a particular domain. Metrics for OO design have suffered from many types of criticism, from lacking a theoretical basis, missing the measurement goals, misleading use when deriving the metric, to simply a metric derivation collection which is too fastidious [ChiKem94]. The results obtained from metrics derived on both C++ and Smalltalk applications [ChiKem94, LorKid94] showed that interpretation of data are usually consistent across the same language. It is suggested that metric results exhibit "typical" syntactic language construct profiles dependent on the language used. This observed fact constitutes one of the main motivations behind the desire of generating a redefinition profile for inheritance hierarchies. Also, such comparison methods could be categorised in the benchmarking technique whereby a chosen set of measures is arbitrarily the reference and where measures obtained from others systems are compared against one or many references.

Lorenz and Kidd [LorKid94] preferred the use of thresholds for their proposed metrics. Thresholds are also arbitrarily chosen numbers for which a measure is believed to be fair. The usual form of a threshold is an average, a minimum or a maximum. Still, in this case, the decision on the validity of a design relies on the comparison of a value obtained against such threshold. Thus, it is arguable why a metric applied in one context should be the reference for the same metric applied in a different context. For example, it is irrelevant that all classes in a model should have the same number of methods as the average case. Such comparisons might only hold in the case of two or more similar classes representing a slight variation of an abstraction.

One possible approach to tackle the problem of interpretation is in the understanding of the dependencies between object concepts. As the metrics are applied on the internal features of an object model, it is interesting to investigate how dependent the metrics are. The next section investigates such approach and gives insights on the possible interactions between related metrics.

#### 2.4.6.1. Remark on the dependencies between metrics

Unsurprisingly, in object technology as in many other technologies, concepts are directly or indirectly related to each other. The notion of relationship relates to the dependency criteria. Here, the notion of dependency can be defined as follows:

---

##### **Dependency between metrics**

A metric  $m_1$  is dependent on a metric  $m_2$  if and only if there exists a characteristic  $c$  which affects the values of  $m_1$  and also affects the values of  $m_2$ .

---

In general, objects that exchange messages are dependent on each other. In the literature, only a few experiments with metrics for object-oriented systems emphasise this dependency aspect between the concepts measured [Ban97, HitMon95a]. It is argued that a dependency between metrics also exists if the respective attributes measured are dependent on each other. Therefore, it would be possible to exploit such a property to support and facilitate the use and interpretation of metric results. Based on the knowledge of the dependency factor between metrics, one possible investigation technique would be to simulate a set of results for one metric and infer the results for others. Thus, inference of the corresponding design may be predicted.

In a measurement programme, it is common to use a set of metrics rather than a single one. The reason lies in the interpretation of the results and feedback for the designers. Usually, the results of a single metric are not beneficial if considered alone. Adopting a comparative approach permits drawing conclusions relative to a known entity. Thus, knowing the dependencies between metrics would facilitate the interpretation of the results. Indeed, it is not predictable how a metric behaves when derived over a set of applications or even on different versions of the same application. However, the rules for interpretation of metric results should remain consistent with the original assumptions and hypothesis described during the metrics definition phase. For instance, in [Hen96], for the Reuse ratio  $U$  and the Specialisation ratio  $S$  metrics (see section 5.6), the following interpretation values were given:

	<b>Deep hierarchy</b>	<b>Wide hierarchy</b>
<b>U</b>	1-	0
<b>S</b>	1+	$\infty$

The **Reuse ratio** indicates how inheritance of classes is used. The value obtained is less than 1 but if it is near 0, it indicates a shallow, broad hierarchy. The **Specialisation ratio** gives indication about the width of the hierarchy. For a broad structure,  $S \gg 1$ , and for lots of multiple inheritance,  $S \ll 1$ .

Thus, the prediction of evolution of a desired characteristic may benefit from the knowledge of the dependency factors between metrics. Although finding dependencies between metrics constitutes

another topic of research and out with the scope of this thesis, it is discussed as further work in chapter 6.

Chapter 3 explains how the use of inheritance in class hierarchies can generate complex design situations which affect the future of the hierarchy. In particular, the detailed study of the method redefinition mechanism unveils previously unknown design situations that raise issues on the overall quality of the design solution. The reasons why such situations are considered as bad design practices are given, thereby permitting the description of a new heuristic for the identified problem. In order to assess the behavioural inheritance aspect of a design, the design factors that influence the design process are reviewed together with the possible forms of method redefinition. Then, a novel set of metrics is proposed to tackle the identified problem. Finally, a data interpretation technique is presented and addresses the issue of analysis of metrics results.

### **3. Assessing the Properties Inheritance Scheme for the Multiple Descendant Redefinition Problem in Object-Oriented Systems**

*"The purpose of abstraction is to separate behaviour from implementation"*

– Barbara Liskov [LisGut86]

#### **Object-oriented design and assessment model: a refocus on the designer**

To date, the area of measurement for OO systems has mainly focused on internal characteristics of the design such as the number of classes, the number of messages sent and received by a class or the depth of inheritance. Although these characteristics enable the definition of metrics, this section emphasises the fact that a refocus on the goals definition phase is needed. An assessment process should be design-driven and design-centered rather than being metric-centered as is often the case. If an assessment of an object model is desired, the detection of the pertinent internal characteristics does not suffice. The definition of the goals of measurement is highly dependent on the context of the measurement. In Figure 3.1, an OO design assessment model describes the main actors participating and influencing the result of an assessment programme. This is often omitted in the literature. It is believed that this is one of the main reasons why metrics are potentially misleading.

To assess software applications, there are three main aspects to consider which are materialised as a three-layer model shown in Figure 3.1:

1. The object-oriented fundamentals.
2. The human factors.
3. The software development processes and products.

The representation of the three layer object-oriented design assessment (OODA) model in Figure 3.1 principally shows the relationships involved between the major actors of a design process and the processes themselves. The presence of human factors in the middle layer of the model emphasises the fact that the role of the designer is the central key to the development. Indeed many automated tools such as diagramming tools and code generators are helpful tool aids in the design process, but these remain limited to a set of functionalities where the interaction with the designer is still required. Similarly, for the interpretation process, the decision-making and the conclusions are, in general, drawn by the designers. Otherwise, if defined and precise interpretation rules exists, tools may be able to handle them and infer the corresponding conclusions.

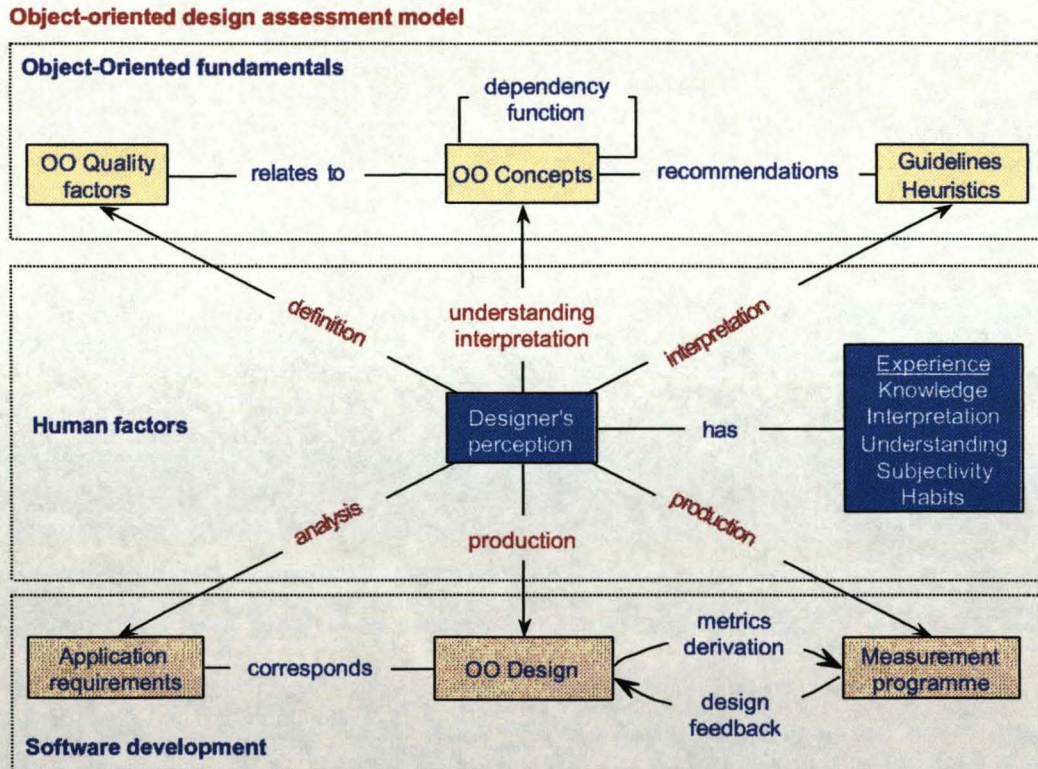


Figure 3.1: Object-oriented design assessment model

Figure 3.1 shows the interactions of the different components involved in the software development process. The first layer is concerned with the fundamental object-oriented concepts. In this layer, the *Guidelines/Heuristics* component remains one of the most intuitive and practical techniques for understanding and using object concepts (see section 2.3). During an assessment programme, the main goal is to quantify the level of “goodness” or “badness” of the characteristic measured. In relation to these defined criteria, a set of reference values i.e. threshold values delimiting the “good” from the “bad”, are usually needed when the purpose of the assessment is to compare results of the same metrics on several parts of the design. The *dependency function* relationship on the *OO concepts* component notifies the fact that an implicit dependency factor ties concepts together. When the designer is able to capture and understand such dependency factors, the interpretation of metric results is facilitated.

The middle layer relates to the human factor issues in the process of designing and assessing. Although all design problems imply different design solutions, there are approaches to recognise “reusable design chunks” i.e. design patterns [Gam&a195] because of the similar nature of the problems. The designers judgements and choices are dependent on their own experience and perception of the concepts. The experience of the designer is shown as a list of features including the knowledge, the interpretation, the understanding, the level of subjectivity and the habits. All of these features play an important role in the success and correctness of interpretation of metric results. If it was intended for a design to have a particular structural and behavioural organisation, the assessment of the design will indeed reflect this desire in terms of quality criteria. Overall,

conclusions are satisfactory when solutions for design improvement can be obtained from the assessment of a particular characteristic of design.

The third layer concerns the software design process. In Figure 3.1, the *application requirements* and the *measurement programme* components are included as part of the software development layer. The *OO design* component outlines the fact that the OO model produced is subject to a measurement programme providing that the necessary design information for the derivation of metrics is available and valid at this time. In chapter 6, a proposed model for the integration of the measurement programme within the design process is discussed.

The OODA model emphasises the important role of the designer both in the design and assessment processes. Moreover, it shows the various tools available to aid the designers when considering the evaluation of an OO design. In order to obtain accurate and useful conclusions from an assessment programme, it is necessary to reduce the number of factors which cannot be quantified, especially when related to the designer's perception.

### **Motivation**

Given the OODA model, it is clear that the production of an object model depends on the designer's interpretation and understanding of the object concepts. A possible approach to evaluate the goodness of an object model is to validate it against suitable design guidelines. This chapter concerns the study of the inheritance mechanism and the effects expected and produced in a hierarchy of classes for object-oriented information systems. The reasons why complete method redefinition infringes the essence of inheritance are discussed. To do so, the design methodology issues concerning behavioural inheritance are examined. A redefinition metrics set is proposed and practical experiments demonstrate that the results obtained permit the detection of inheritance design problems. Appropriate design decisions are suggested.

This work aims at a comprehensive analysis of the extent of the redefinition mechanism using metrics for object-oriented systems in order to identify a simple methodological approach to the problem of measurement. It is also aimed at providing guidance as to the appropriate use of redefinition for improvement of behavioural and conceptual properties of the model. The information gathered from the metrics is then used in a design-evaluation cycle.

The key contributions are:

- An identification of design methodology considerations related to inheritance assessment.
- An identification of design inconsistencies resulting from the multiple method redefinition problem in a class hierarchy.
- The proposition of a method redefinition metrics set for assessing inheritance from a behavioural viewpoint.

- Empirical validation of the metrics set and results obtained from the Smalltalk class library are presented.

The next section explains how and why, in some situations, method redefinitions can severely compromise the reusability and maintainability of the model. In section 3.2, a redefinition metrics set is proposed and aims at measuring redefinition activity in class hierarchies. Section 3.3 provides a methodological approach where further design issues are examined regarding the assessment of method redefinition in class hierarchies. Finally, in section 3.4, a data interpretation method is proposed for addressing the problem extraction of feedback from the analysis of the metrics results.

### 3.1. Method redefinition: uses and abuses

Current use of inheritance has illustrated that the introduction of conceptual inconsistencies is possible in a class hierarchy. Based on the analysis of current existing class hierarchies, potential design problems may arise in an object model due to an unclear use of the method redefinition techniques. Languages are fundamentally different as each provides different ways of implementing OO principles such as encapsulation or method redefinition; thus this implementation has close equivalents in other languages. As the focus is given to Smalltalk's implementation of the redefinition concept, it is important to note that such implementation has its equivalent in other languages; therefore the analysis presented here also applies to other languages. The context of the problem is outlined and a heuristic is created to capture its essence. It is explained why such redefinition uses pose major issues for the future maintenance of the hierarchy. Thereby, the problem's definition sets the scene for the remaining part of the thesis and serves as the basis for the evaluation of goodness of inheritance hierarchies.

#### 3.1.1. Method redefinition in class hierarchies

A major criticism of redefinition lies in the essence of inheritance itself. The two notions of property redefinition and property heritage are paradoxical. Surprisingly enough, method redefinition, including correct and incorrect use, happens more often than expected in a class hierarchy. For example, the redefinition metric results for the Smalltalk class library (Figure 3.2) show that the amount of redefinition reaches 57.07% at DIT=4 in the hierarchy. On the first three levels of the hierarchy, the results obtained more than double from one level to another, denoting high "redefinition activity". One possible reason for such a redefinition profile is the incremental development of software. A closer look at the implementation of the same method redefined many times along a branch of the hierarchy revealed that common code had not been factorised. This phenomenon seems typical of the case of many developers working on the same part of a system without modifying the others' code (class dependency problem). Chidamber and Kemerer's *coupling between objects* (CBO) metric [ChiKem94] permits the detection of weak and strong coupling. The CBO is recommended to be as low as possible. However, with new design

techniques such as design patterns [Gam&a195], the dependency between classes present in a pattern is high as they are strongly dependent (the purpose of a pattern).

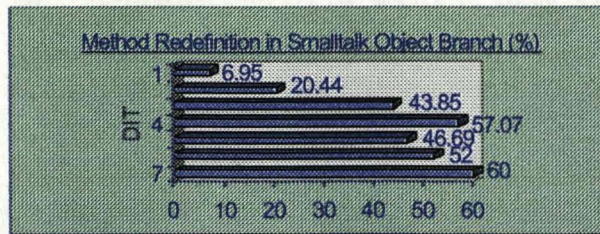


Figure 3.2: Smalltalk hierarchy redefinition profile

Smalltalk has been criticised for its implementation inheritance [Rum91, Tai96]. For instance, cancellation, which is a variant of implementation inheritance, is common in the class hierarchy. Similarly, Bracha and Cook [BraCoo90] stated that inheritance in Smalltalk is a mechanism for incremental programming whereby instances of a class may not bear a necessary relationship with the instances of its subclasses. Again, inheritance is used for convenience reasons and behavioural compatibility may be ignored. Nonetheless, Taivalsaari [Tai96] acknowledged that the Smalltalk class hierarchy has its advantages. It is generally recognised that the hierarchy would be more complex and memory consuming if it was designed in a more conceptual approach. Cook [Coo92] described some major problems in the Smalltalk hierarchy as follows:

- Inherited methods that violate the subclass invariant.
- Methods that have the same name but completely unrelated behaviours and for which a generalised specification cannot be found.
- Methods that have the same (or related) behaviour but different names.

All the above-mentioned problems contribute to the introduction of potential design inconsistencies such as the MDR problem in the class hierarchy. The next section formalises the unusual case of method redefinition and explains why it is conceptually wrong.

### 3.1.2. Multiple descendant redefinition (MDR) problem

The principle of inheritance involves an ownership transfer of features from the parent class to its subclasses. When a class inherits a method which has been publicly defined, the subclass has the right to change the property inheritance scheme for itself and future heirs.



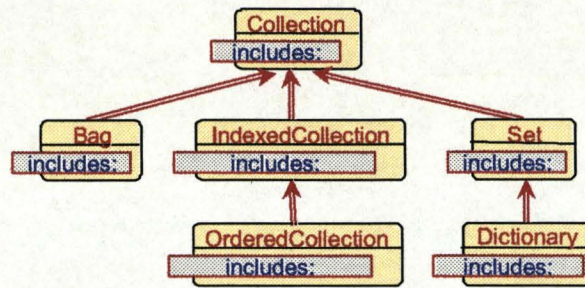


Figure 3.3: Life history of the includes: redefined method in the Smalltalk Collection branch

In Figure 3.3, the includes: method is used to test if an element is present in a collection. At first sight, a representation of the life history of the completely redefined includes: method casts doubt on the correctness of the design. Although all IndexedCollections are Collections, they do not test the inclusion of elements in the same manner, as IndexedCollection introduces a key for access. The solution is thus to redefine the includes: method to cancel the inherited implementation from the class Collection. Similarly, for OrderedCollection, the same method is completely redefined again. Clearly, the property inheritance scheme is broken and nothing is inherited from the parent class. Furthermore, the includes: method has not been originally declared as deferred and all its subclasses hold completely different forms, an incorrect case of polymorphism by definition. This situation will be referred to as the *multiple descendant redefinition* (MDR) problem. It should be noted that such classification, although conceptually incorrect can be implemented in any programming language. Further complex method redefinition situations may also arise when a combination of many super calls exists in the same method.

A definition of MDR is as follows:

---

In a class hierarchy, consider a class  $\text{parentC} = \{ \langle \text{mthA}() \rangle \}$  and  $\text{mthA}()$  declared as public.

$$\text{MDR} \exists \text{ iff } \left\{ \begin{array}{l} \bullet \{ \forall \text{subclassD}, \forall \text{subclassE} \mid \text{subclassD} < \text{parentC}, \text{subclassE} <_{\text{direct}} \text{subclassD} \} \\ \bullet \text{subclassD} = \{ \langle \langle \text{mthA}() \rangle \rangle \}, \text{mthA}() \text{ is replaced} \\ \bullet \text{subclassE} = \{ \langle \langle \text{mthA}() \rangle \rangle \}, \text{mthA}() \text{ is replaced} \\ \bullet \text{mthA}()_{\text{subclassD}} \neq \text{mthA}()_{\text{subclassE}} \neq \text{mthA}()_{\text{parentC}} \end{array} \right.$$

where the relation  $\text{classB} <_{\text{direct}} \text{classA}$  denotes the fact that classB is a direct subclass of classA and  $\text{mth}()_{\text{classA}}$  is read as the method  $\text{mth}()$  of classA

---

To illustrate how MDR problems can be tackled in class hierarchies, an example of an alternative design solution is given in the next section.

### 3.1.3. Example inheritance hierarchy that avoids the MDR problem

Although the study of solutions to the MDR problems is outwith the scope of this thesis, suggestions for improvement of a class hierarchy are presented in this section.

Inheritance hierarchies that encompass MDR problems require a re-design of the hierarchies which usually implies code re-engineering. Many viable solutions are possible to tackle the MDR

problem; however it is important to emphasise that they are not straightforward as other related design aspects have to be considered. For instance, if an alternative solution consists in moving a method *M* from a class *A* to a class *B*, the consequences of such relocation have to be examined. As the original property inheritance scheme is affected, subclasses of *A* may still expect the inheritance of method *M*. In general, the presence of MDR problems in a hierarchy indicates a more broader design problem. Note that potential solutions to the MDR problem also depend on the language features. To address the problems of the Smalltalk hierarchy mentioned in section 3.1.1, Cook proposed an alternative *Collection* class hierarchy based on the conceptual relationships of the classes [Coo92]. He demonstrated the use of interface hierarchies and specification techniques in producing an improved class library structure. Bracha and Cook [BraCoo90] proposed the concept of *mixin-based inheritance* as a new inheritance model. The model relies on composition of *mixins* or abstract subclasses. Separate mixin classes are created to hold parts of classes that may not be related but sharing a set of common behaviours. In that respect, mixin classes seem a good candidate for solving the MDR problem. Both techniques of interface hierarchies and mixin-based inheritance constitute potential candidates to avoid MDR problems. The latter is used in the example below.

As Smalltalk supports single inheritance, one of the main problems of its class hierarchy is that code may be duplicated across different classes and by side effect this situation often generates MDR problems.

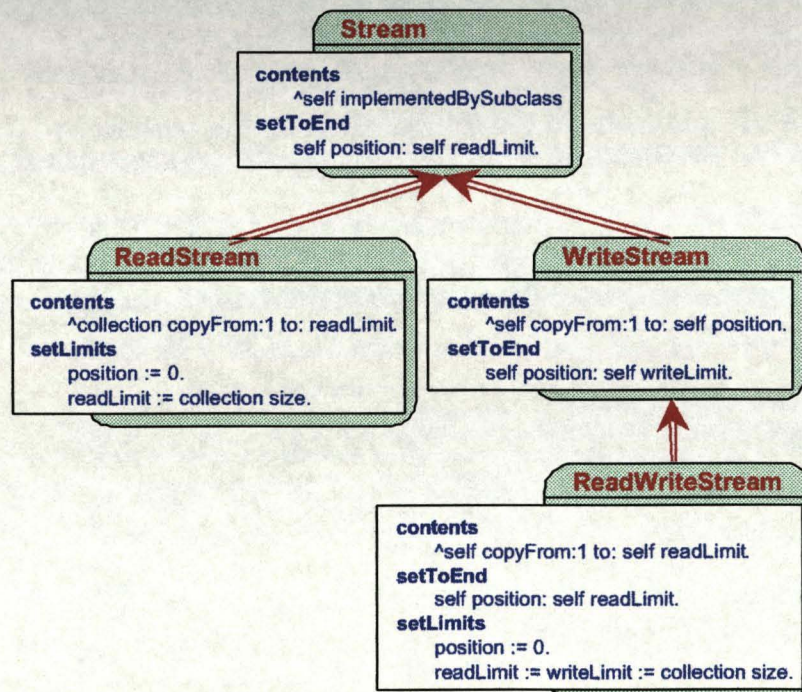


Figure 3.4: MDR and code duplication in the *Stream* class hierarchy

Figure 3.4 shows the Smalltalk Stream hierarchy which includes two main problems as follows:

- Code duplication due to Smalltalk's single inheritance. The `ReadStream` class only inherits from the `WriteStream` class but as behaviours of the `ReadStream` class are needed, duplication of the `setLimits` method is done.
- Presence of MDR due to the non-full compatibility between `ReadStream` and `WriteStream`. Strangely enough, the `setToEnd` method is originally declared in the `Stream` class although the definition of its body appears to be for the `ReadStream` class. `WriteStream` completely redefines the method and so does `ReadStream`, giving rise to the presence of MDR. Note that the body of the `setToEnd` method in `ReadStream` is the same as the one originally defined in the `Stream` class. This situation illustrates a case of use of inheritance for convenience reasons. Originally declared as abstract in `Stream`, the `contents` method in `ReadStream` is also suspect as its body is very similar to the one in `ReadStream`.

In this particular example, note that multiple inheritance as described in section 2.1.8 represents a possible solution to the code duplication and MDR problems. However, in the alternative design solution below, the use of mixins is presented<sup>13</sup>. It is believed that mixins represent a better solution to tackle MDR problems in a wider context.

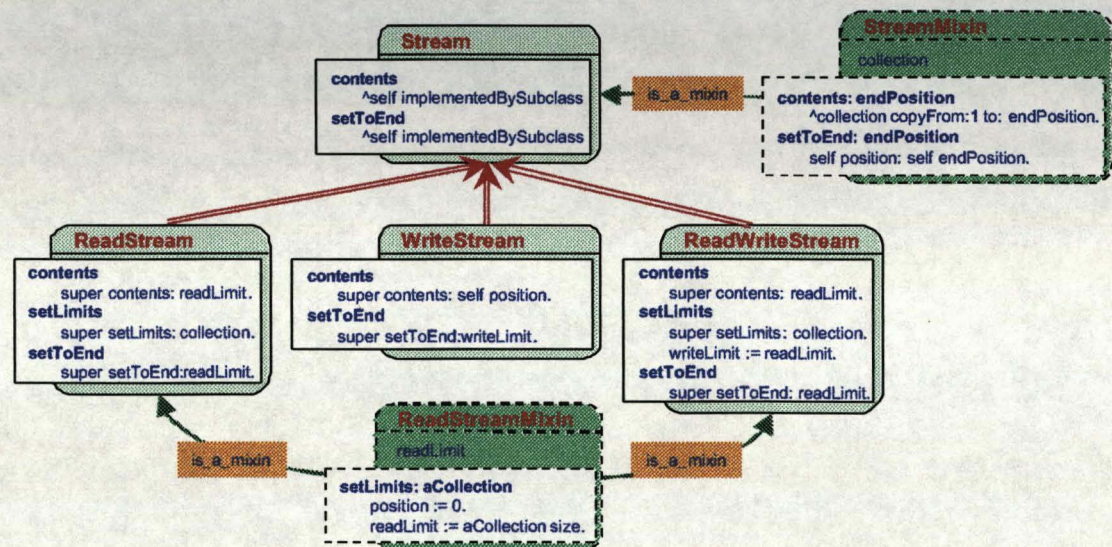


Figure 3.5: Stream hierarchy using mixins classes

Figure 3.5 shows an alternative Stream hierarchy which introduces two mixins classes: `StreamMixin` and `ReadStreamMixin`. Design solutions using native Smalltalk capabilities may be found for simulating mixins, however the model would probably be simpler with the use of real mixins. In this example, it is assumed that Smalltalk has been extended to include mixins capabilities as described in [BraCoo90] or [Sch98]. The introduction of mixin classes captures

<sup>13</sup> See [BraCoo90] for the details of mixins' implementation.

common behaviours in the hierarchy; however existing classes also need to be altered so that methods are still accessible. To do so, the `Stream` class is combined with the `StreamMixin` class, `ReadStream` and `ReadStream` class with `ReadStreamMixin`. In the `Stream` class, the `contents` and `setToEnd` methods are declared as abstract methods, thus encouraging the use of polymorphism. `ReadStream` is now treated as another type of `Stream` for the reasons that it still inherits the common behaviours from the `Stream` class but can also be combined with the mixin classes so that specific behaviours to the `ReadStream` and `WriteStream` classes are available. All subclasses of `Stream` make use of inheritance for extension and both code duplication and MDR problems are avoided.

Mixin classes appear to be a good candidate for tackling implementation inheritance; however the cost of a re-engineering process should not be underestimated. Although the alternative design is conceptually sound, the increase in complexity and amount of code is noticeable.

The next section illustrates the consequences of a MDR problem regarding the property inheritance scheme.

#### 3.1.4. Descendants heritage extent for the MDR problem

In an extreme situation, suppose that the `Parent` class completely redefines all the `Grand-parent`'s methods, and the `Child` class redefines all the `Parent`'s methods: all versions of the methods defined in the `Parent` and `Grand-parent` classes are lost (Figure 3.6). In the `Child` class, no features come from its ancestors although being a subclass.

---

An MDR heuristic can be formulated as follows:

Providing the hypothesis that the multiple descendant redefinition problem breaks the properties inheritance scheme in a class hierarchy, a method `m` from a class `C` should not be consecutively and completely redefined more than twice down a given branch. If such a situation occurs, all versions of method `m` defined in previous ancestors classes are lost, thus violating the essence of inheritance.

---

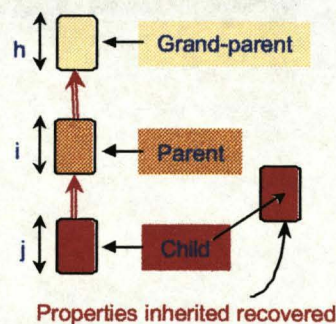


Figure 3.6: Descendant heritage extent with MDR anomaly.

### 3.2. Measuring redefinition in object-oriented systems

The method redefinition mechanism can be applied in obscure manner in class hierarchies and is not always justified [Mey88, Rum91]. In a parent-child relationship between two classes, the shared methods are the ones defined in the parent class as inheritance is unidirectional. In 3.1.1, it was shown that a high rate of method redefinition occurs in the current Smalltalk class hierarchy and that such a situation may point to potential design inconsistencies as methods are generally expected to be extended rather than being redefined. A high rate of completely redefined methods strongly suggests some behavioural inheritance design problems e.g. MDR problem. This might indicate that either the parent class has poorly abstracted the methods concerned or the subclasses are wrongly situated in the hierarchy which obliges the class to ignore inherited properties. On many occasions, a deep analysis of the class hierarchy source code depicted that suspect methods can simply lack code factorisation and thereby fall under the case of a complete redefinition instead of an expected extension. It was suggested that, due to the class dependency problem and the incremental software development, developers would prefer to re-write their own version.

Given the MDR heuristic (section 3.1.2) and the design considerations for inheritance assessment (section 3.2), it is now possible to elaborate a measurement plan that specifically tackle the MDR problem. The following sections describe the application of the GQM/MEDEA model for building a redefinition metrics set.

#### 3.2.1. The method redefinition assessment

Current criticisms of OO metrics are that they only provide hints or clues to the “goodness” of the design. We argue that a precise identification of suspected problems with valid metrics for its assessment suggests obvious directions or solutions for design improvement. With the help of the behavioural analysis technique (section 3.3.5), metrics can be prescriptive.

The approach taken to define the product metrics was based on GQM/MEDEA (Goal Question Metric/Metric Definition Approach [Bas92, Bri&al94]) which provides practical guidelines for building metric sets. Nonetheless, this stage remains a difficult process for determining the validity of the metric. Whilst Ebert stated that “*a metric is a criterion to determine the difference or distance between two entities*” [Ebe92], the definition of the criterion itself is subject to difficulties. Many metrics design models have refined the process by which less uncertainty is allowed regarding the definition of objectives for a metric. Thus, the very first step in defining a metric is the “Experimental goal(s) definition” stage, defined as the set of the following topics [Bri&al94].

The steps involved in applying the method are:

**Step 1: Experimental goal(s)**

*Object of study:* method redefinition mechanism in a class hierarchy

*Purpose:* detection of MDR anomaly

*Quality focus:* conceptual design consistency for property heritage

*Viewpoint:* designer

**Step 2: Assumptions**

*Assumption 1:* the deeper a class is in a tree hierarchy, the more complex it is

*Assumption 2:* the deeper a class is in a tree hierarchy, the more likely the MDR problem arises

*Assumption 3:* see the MDR guideline formulated in section 3.1.2.

**Step 3 and 4:** Relevant measurement concept and product abstractions. The rationale behind the redefinition metrics set is fairly straightforward and has been emphasised in the fundamental steps 1, 5 and 6. The abstract properties of the redefinition metrics are discussed in section 5.10.

**Step 5:** Define the candidate metrics (see section 3.2.2)

**Step 6:** Experimental validation of the metrics (see chapter 5)

A precise definition of the goals reduces the chances for the future metric to be incorrect. Brito et al. [Bri&a194] established that this stage is fundamental to the whole metric definition process. A possible means for identification of goals can be tackled in looking at design recommendations or guidelines. However, in practice, the application of guidelines or heuristics, often in a textual form [Fir95, Mey88, Rie96, Rum91], is not very easy to accomplish (see section 2.3.3).

Again, the quality of the OO model is completely dependent on the designer's experience, understanding and interpretation of the concepts used. At least, guidelines provide a method for recognising good OO design standards.

The following redefinition metrics are proposed and explained in the next section:

- **PRM:** the percentage of redefined methods includes 1) the methods completely redefined, 2) extended and 3) realised (see section 2.2.4).
- **PRMH:** the percentage of redefined methods per level within a hierarchy and its variants (PCRM and PEM)
- **PCRM:** the percentage of completely redefined methods. This metric is intended to assess the first and third cases above mentioned.
- **PEM:** the percentage of extended methods. This is the second case of redefinition.

### 3.2.2. Percentage of redefined methods per level within a hierarchy (PRMH)

Current metrics assessing inheritance examine single classes or a system whereas the PRMH metric evaluates the amount of redefinition level by level. Providing that a class hierarchy is ideally designed, abstract classes should appear closer to the root of the hierarchy and specialised (or concrete) classes should be situated nearer to the bottom. The redefinition metrics are aimed at depicting such a profile. For instance,  $PRMH_1$  metric (Figure 3.7 branch A at level 1) measures the shaded classes. The PRMH metric can also be applied at the system level as classes are not necessarily organised in a class hierarchy. For simplicity, we will keep the numbering level absolute in comparison with the root (class Object) level 0. The notation  $C_{m,n}$  gives the location of a class C, at rank n, for a given level m in the branch, e.g. class B at level 2 of branch A, is named  $B_{2,1}$ . The rank is arbitrarily numbered from 0 to n, n is an integer, from left to right at the considered level. Note that the rank is used only for a logical identification of the classes at a specific level in the formulas below, but does not imply a notion of ordering in the class hierarchy.

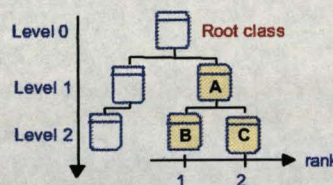


Figure 3.7: Complexity metrics at hierarchy level

The redefinition metric for a class and for a given level m are defined as:

$PRMC = \frac{NRM}{NIM} * 100$	$PRMC' = \frac{NRM}{NPIM} * 100$	$PRMH_m = \frac{\sum_{n=1}^{NC} PRMC_{m,n}}{NC}$	(a)
--------------------------------	----------------------------------	--	-----

where NRM is the number of redefined methods, NIM is the number of instance methods,  $NIM > 0$ <sup>14</sup>, NC is the number of classes for a given level m,  $NC > 0$ ,  $PRMC_{m,n}$  is the percentage of redefined methods for all classes  $C_{m,n}$ . In the current calculation of PRMC (first approach), the equation is a function of the NIM defined locally. However, any class C inherits methods from all its parents, making them potentially available for use (via the method lookup mechanism). For this reason, the *cumulative redefinition approach* to the same calculation is given by the  $PRMC^1$  equation (second approach) where NPIM is the number of potential instance methods,  $NPIM > 0$ . Indeed, NPIM is expected to increase from top to bottom of a hierarchy, thus, PRMH decreases when DIT increases. This metric relates to the fact that “off-the-shelf” class hierarchies are

<sup>14</sup> Note that classes without methods (e.g. classes that defines constants only) may exist but are not relevant in this thesis.

abstracted enough to contain a fairly high depth of inheritance and a high number of methods per class. Consequently, the deeper the class is in the hierarchy, the more it is likely to inherit a high number of methods. Thus, designers face the problem of finding the wanted information amongst a high proportion of non-relevant ones. The PRMH in (a) is general. A refined version includes the redefinition variants:

$PCRM = \frac{NCRM}{NIM} * 100$	$PEM = \frac{NEM}{NIM} * 100$	$PRMH_m = \frac{\sum_{n=1}^{NC} (PCRM + PEM)}{NC}$	(b)
---------------------------------	-------------------------------	--	-----

where  $NIM > 0$ ,  $NC > 0$ ,  $NCRM$  is the number of completely redefined methods and  $NEM$  is the number of extended methods.

In general, the interpretation of the redefinition metrics needs to be done in connexion with other related metrics. For example, consider a class that does not hold redefined methods. The interpretation is likely to be different depending on the total number of methods in the class.

Due to the inclusion of the DIT metric within the redefinition metric set, the depiction of *redefinition profiles* of hierarchies is possible.

In order to detect and thus assess potential design problems such as the MDR problem in a class hierarchy, it is necessary to identify the main design aspects that should be considered in a measurement programme. The conceptual and technical issues involved in such an assessment are explained in the next sections. In particular, it is shown how a state transition diagram describing the method redefinition states permits the identification of the suspect state transitions e.g. the MDR problem. A behavioural inheritance analysis is proposed to tackle the problem of localisation of defective classes in class hierarchies.

### 3.3. Design considerations for inheritance assessment

*“Designing is weighing alternatives, including discovering them in the first place and eventually rejecting all but one” – Chamond Liu [Liu96]*

The MDR problem and the redefinition metrics have been described in the previous section, and contribute towards the goals of a measurement plan. This section is concerned with the description of the technical issues involved in the assessment of inheritance hierarchies and thereby the assessment of the redefinition mechanism. Note that the following mainly constitutes a design exercise which is directly relevant to the essential aspects of assessment. In order to identify a methodological approach in a design assessment activity, four categories of design information are considered:

- The key mechanics for extracting design information from an inheritance hierarchy.
- The definition of the possible method redefinition statuses. This addresses the different type of methods to assess, and thus a possible direction for finding appropriate metrics.



- An essential behavioural inheritance analysis model which enables the designer to focus on a particular branch of a hierarchy. An overview of a branch restricted to the desired methods permits a rapid localisation of suspect classes. This is aimed at supporting the interpretation of the metric results.
- Specific remarks on the consequences of use of method redefinition to be taken into account during analysis of the metric results.

### 3.3.1. Methodological approach for class hierarchy assessment

From a software engineering point of view, satisfying all the requirements for the system is a requirement but achieving a maintainable, flexible and open architecture is as important if it is to achieve reusability with reduction of costs for future development. To date, mechanisms in the object model do not permit full control of the property inheritance scheme [Sei96, Tai96]. In an inheritance hierarchy, the number of features of a class and the number of levels of depth are difficult to manage. In class hierarchies such as the OWL, it is not surprising to have a large number of methods in leaf classes. Note that this may have been what was originally intended. However, when extension or reuse is wanted, such situations rapidly become a burden for the designer because of the exhaustive search process for the existence and origin of desired method's interfaces and implementation. The techniques proposed in the following sections contribute to the detection of possible design problems appearing in class hierarchies. For example, the problem of MDR is effectively seen as a side effect of the use of inheritance. In order to tackle the variety and combination of property inheritance schemes in an object model, it is necessary to be able to assess methods of a class, at any level of the hierarchy. As a complementary tool for the designer, the techniques address the reuse or extension of a class hierarchy from a behavioural point of view. To help designers in pinpointing design defects, the following design methodology approaches are considered:

- **Behavioural inheritance analysis:** in class hierarchies, the transfer of ownership (see section 2.1.6) and the redefinition mechanism (see section 2.2.1) constantly change the state and definition of the original method. In order to have an overview of the history of a particular method in a class hierarchy, the creation of a *method's life history* record enables the discovery of the origin and successive definitions of the method.
- **The definition of a metric set:** the use of a set of redefinition metrics applied to a branch of the hierarchy or the whole hierarchy would permit the representation of the notion of redefinition profiles. One possible way to assess the amount of methods redefined is to isolate branches within the hierarchy. Particularly, in single-rooted object-oriented systems, the abstractions are derived from the same root class, therefore the only possible way to isolate them is to consider the start of a branch at a defined node. Then, on a graphical representation, a depiction of the

redefinition profile would help in the understanding of the general evolution of the property redefined.

- **Interpretation:** the identification of potential defect classes can be done using a cross-reference method between the interpretation metric results and the method's life history technique. Although a redefinition profile may already indicate potential design problems in a class hierarchy, the precise localisation of a design defect requires the support of additional method analysis tools described in the section 3.2. Possible useful processing tasks may involve filtering, graphical representations and data mining.

The next section explores the technical aspects that allow the extraction of information from an inheritance structure.

### 3.3.2. A design information repository with metaclass facilities

*"To perform measurements on a program or design, we need to be able to describe the structure of a program or design in language-independent terms." – Anton Eliëns [Eli95]*

This section explains how the extraction of design features is possible using metaclass facilities. Due to the incremental design process, classes and their properties are likely to change during the course of design. The main problem of early measurement relates, not only to the availability of the design information but also to the degree of correctness of the information (see section 2.4.5). Even in the case of use of supporting tools such as diagrammatic or CASE tools, the derivation of metrics implies that metrication functionalities are already implemented within the design tool in order to share the meta-information generated by the design tool [LewSim98]. Measurement techniques may be applied at any time in the development process providing that the required design information is available and consistent.

The following four sub-sections describe the core set of design information that is used within the metric's calculation algorithm. The purpose of a design information repository is to identify all design characteristics relevant to a measurement process.

## Meta-model

Consider the following meta-model which is used to represent the main OO concepts:

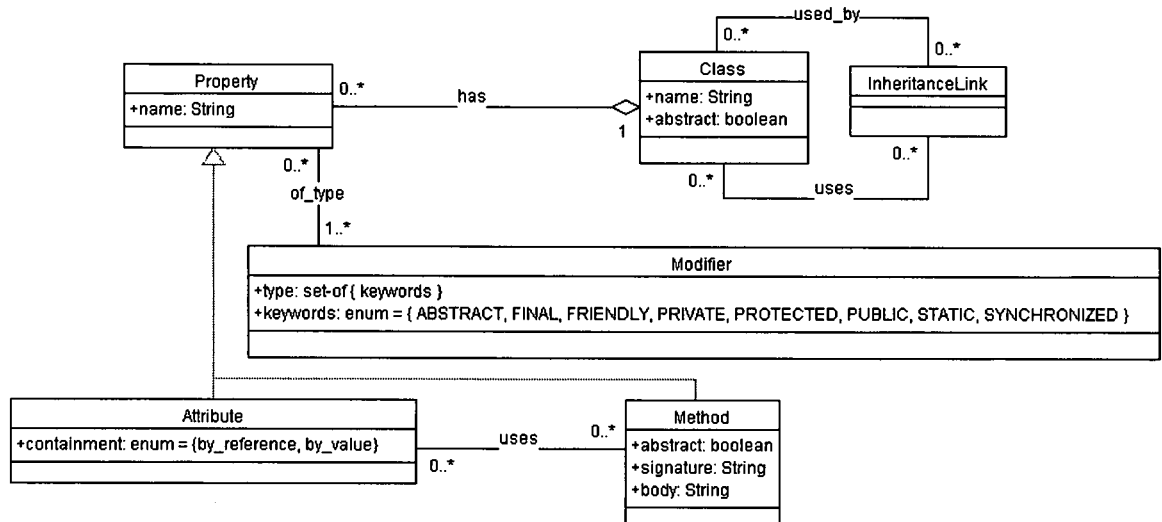


Figure 3.8: Meta-model of main OO concepts

In Figure 3.8, classes and properties i.e. attributes or methods, are modelled as classes. Instances of the class **Class** have instances of the class **Property**. In a class, the relationships with other classes are defined by constructing new instances of the other class. For this reason, relationships can be modelled as instances of the class **Property** and act as aggregates of instances of the class **Class**. The type of a **Property** object is defined by a possible combination of **Modifier** objects. The **Attribute** class and the **Method** class both inherit from the **Property** class. Relationships between classes can simply be categorised in two groups: the inheritance relationship and all other types of relationships. Indeed, the latter category can be subdivided in many more groups to differentiate from a simple association, aggregation, dependency, etc. The containment attribute in class **Attribute** notifies the fact that an instance attribute can be attached either as a nested component/composite objects or as a pointer to a composite object. Another possible way to describe a relationship between two classes can be done within the body of a method. Local variables to the method can be temporarily declared of a particular class type (section 2.1.7).

In the meta-model presented above, the interesting design features are the class properties. Clearly, each of them is a potential metric. For example, for a class, “*the number of methods per class*” can be calculated in counting the number of the **Method** class’s instances. Thus, the meta-level design information provides a description of all design features which can be used by the metrics’ algorithms.

Note that **Modifier** objects that are incorrect Java modifiers can be defined according to the meta-model. For instance, the value of the variable **type** may be: {**ABSTRACT ABSTRACT**} which is an incorrect Java modifier. A semantic analyser or improved meta-model can detect such error cases. The purpose of the meta-model is to show how the capture of meta-information can be done.

### Remarks on the encapsulation aspect

In Figure 3.8, the attribute `scopeModifier` of the `Property` class gives the indication of the property's visibility for the heir classes i.e. encapsulation. The C++'s `PUBLIC`, `PRIVATE` and `PROTECTED` scope modifiers are the ones described in section 2.1.7. The `FINAL` property modifier gives visibility of the property to heir classes but prohibits its redefinition. It is equivalent to a removal of the property. A peculiarity of the inheritance relationship is that it has been separately modelled with a self-link, via the `InheritanceLink` class, on the class `Class`. The reason lies in the semantics of inheritance. An inheritance relationship implies a transitive transfer of the properties from the parent to the child class. It purely deals with the behavioural aspect of two classes: one is able to use and modify properties from the other one. As opposed to other relationships, the inheritance relationship acts as a channel for ancestor's property visibility where the other relationships are mainly resulting from the declaration of variables in a class. It is basically the use of the two groups of relationships which combines classes together and communicates via message-passing that provides the expressiveness of the OO concepts. It is the combination of different property scopes in a class hierarchy which is essentially responsible for the complexity of the inheritance scoping control. By consequence, the validity and correctness of the design is also affected by the property scope modifiers.

Figure 3.8 illustrated some of the desired design features that can be used for the computation of metrics. As these metrics would constitute the basic metrics, it is, therefore interesting to build a repository of such metrics based on the collected design information. Indeed, such a repository is convenient for building more complex metrics. This approach will be considered for building a metric collector tool.

### Detecting a method's original definition

Another aspect of the retrieval of design information concerns the identification of the class's context such as its references to internal or inherited properties. In particular, to assess behavioural inheritance, for each class, methods are analysed regarding whether it is a new method for the class or if it is inherited. In some class browsers such as RationalRose98®, a class can optionally display the list of inherited methods as well as the new methods. However, if a method is redefined, its method name, signature and body appear in the class description as if it is an added method. To find out if such a method is extended, cancelled or replaced, a finer analysis of the body of the method is required. For example, if the method reuses inherited methods, calls to the ancestor's method will be tagged with the keyword `super`. Note that, unless there is detailed design documentation, the only way to find such information is unfortunately to wait for the source code availability. Thus, analysing the references made to other methods, within a particular one, will enable a finer assessment of the inheritance model used and potential suspect classes and methods.

### Meta design information

A possible categorisation of useful design information concerning the behavioural assessment of inheritance is given in the following Table 3.1, Table 3.2 and Table 3.3. The main interest of such information gathering will serve both the computation of metrics and the suggestion and localisation of design defects. The designer will rely on the availability of front-end tools to manipulate the information. Examples of front-end tools include a metrics collector, methods profiler or persistent storage tools.

<b>Types of meta information for a class</b>	<b>Characteristics</b>
<u>General</u>	<ul style="list-style-type: none"> <li>• Name</li> <li>• Abstract</li> </ul>
<u>Heritage link</u>	<ul style="list-style-type: none"> <li>• direct parent class(es)</li> <li>• list of ancestor's classes</li> <li>• list of direct sub-classes</li> </ul>
<u>Class internals</u>	<ul style="list-style-type: none"> <li>• list of attributes and related information such as name, type, scope</li> <li>• list of internal, inherited methods and related information such as name, returned object type, signature</li> </ul>

Table 3.1: Class design features

In Table 3.1, description of attributes and methods of a class are included in the list. Note that inherited methods are also listed. Some languages provide method look-up mechanisms to infer the list of all inherited characteristics from ancestors. Either in a designer or from an assessment perspective, it is important to know what a class is i.e. its structure but also what it is capable of i.e. its behaviour, inherited or not. Heritage links are the relationships which attach a (many) parent(s) class(es) to its child classes.

<b>Types of meta information for an attribute</b>	<b>Characteristics</b>
<u>General</u>	<ul style="list-style-type: none"> <li>• name</li> <li>• scope</li> <li>• defined in class</li> </ul>
<u>Category</u>	<ul style="list-style-type: none"> <li>• instance attribute</li> <li>• class attribute</li> </ul>
<u>users of</u>	<ul style="list-style-type: none"> <li>• list of internal methods referring to the attribute</li> </ul>

Table 3.2: Attribute design features

In Table 3.2 the characteristics for attributes are shown. However, as the focus of this chapter is on behavioural inheritance, only the fact that a method uses one or other attribute is of interest.

<b>Types of meta information for a method</b>	<b>Characteristics</b>
<u>General</u>	<ul style="list-style-type: none"> <li>• name</li> <li>• abstract</li> <li>• scope</li> <li>• defined in class</li> </ul>

<u>Category</u>	<ul style="list-style-type: none"> <li>• instance method</li> <li>• class method</li> </ul>
<u>Calls from</u>	<ul style="list-style-type: none"> <li>• list of internal methods calling or using the current method</li> </ul>
<u>Calls to</u>	<ul style="list-style-type: none"> <li>• list of internal methods calls including <i>super</i> calls i.e. inherited methods</li> </ul>
<u>Use of</u>	<ul style="list-style-type: none"> <li>• list of internal or inherited instance attributes used</li> <li>• list of class attributes used</li> </ul>

Table 3.3: Method design features

In Table 3.3, the method's interactions are described. Basically, there are two forms of interaction:

- **calls\_from**: interaction between methods are based on a sender-receiver model. The receiver is able to identify the list of senders.
- **calls\_to**: similarly to **calls\_from**, a method uses other methods as receivers. In this case, messages can also be sent to inherited methods. The method binding mechanism makes sure that the correct method receives the message.

Given the above-described list of meta design information, the calculation of metrics becomes fairly straightforward. The next section describes parsing considerations within a class hierarchy.

### 3.3.3. Class analysers

#### **Inheritance path isolation**

The technical issues involved in the extraction of the design features are covered in this section. To assess behavioural inheritance in a class hierarchy, parsing of a tree is necessary. In addition to the design information described in section 3.3.2, a more detailed analysis of the methods in each class permits the investigation of the *method life cycle* or *life history* down the branches of the tree. The designer will rely on the presence of supporting tools to extract such information. To understand the overall effect of the application of scope modifiers to methods in the hierarchy, an isolation of all possible paths is undertaken. Recall that from a designer's perspective, when (re-)using or extending the class hierarchy, the main problem is to discover and understand the successive versions of the same method, especially for bottom classes. It has been generally recognised that class libraries often encompass more functionalities than an application would really need. Note that this is a desired characteristic for class libraries. However, Hitchens and Firmage [HitFir97] stated that the use of a class library is haphazard. With the absence of browsing, query tools or other mechanisms, the designer must proceed through all the classes with no guarantee of finding the desired class. If addition of new classes is needed in large hierarchies, one of the consequences of the situation described above is that classes tend to ignore all un-wanted methods, therefore, risking non-conformance. Whenever used for pragmatic reasons such as possible savings in code development or optimisation purposes, inheritance becomes questionable.

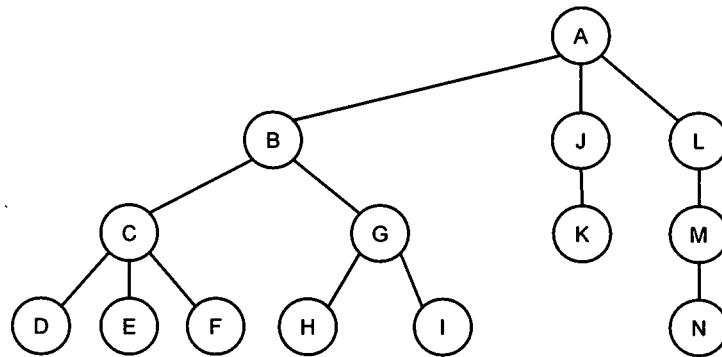


Figure 3.9: Tree parsing strategy

In general, a designer is interested in knowing which are the ancestor's inherited properties. To do so, an isolation of inheritance paths is a possible solution to reveal the desired design information. Figure 3.9 represents an example class hierarchy. Adopting a depth and top-bottom tree parsing strategy, the list of possible paths include:

Level	0	1	2	3
Path 1	A	→ B	→ C	→ D
Path 2	A	→ B	→ C	→ E
Path 3	A	→ B	→ C	→ F
Path 4	A	→ B	→ G	→ H
Path 5	A	→ B	→ G	→ I
Path 6	A	→ J	→ K	
Path 7	A	→ L	→ M	→ N

Table 3.4: Inheritance paths table

The depth and top-bottom (DTB) parsing strategy allows a chronological construction and gathering of design information in the table. Different parsing strategies will be used to examine the behavioural aspects of each of the classes. Note that the parsing strategies mainly concern the issues involved in developing the metric's calculation algorithm; however, it also depends on the encapsulation mechanism in place. Designing and assessing a class hierarchy should really be based on the examination of inheritance paths as a whole. Often, designers only concentrate on direct (or immediate) parent classes to extend the hierarchy instead of inspecting all previous ancestors. The knowledge of chronological changes happening to inherited methods is essential to minimise obscure inheritance uses. Recall that, although not being good practise in a team development, software engineers tend to leave unclear existing pieces of code as they are and redevelop their own version for safety reasons, not encouraging reuse. Often, the fear of modifying someone else's code is not so much due to the code being unclear but due to possible dependencies on other portions of code.

Notice that in the case of multiple inheritance, the detection of the path with a DTB strategy raises the issue of name collisions (see section 2.1.8). Consider the following example (Figure 3.13) where the method `m1()` in class A is publicly inherited in all heir classes B, C, D and E.

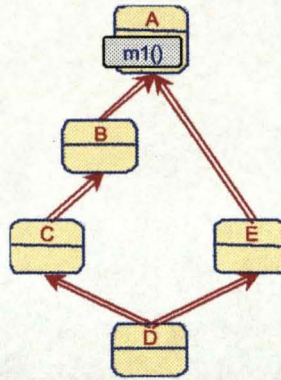


Figure 3.10: Name space collisions with multiple inheritance

Using the DTB strategy, the isolated paths are:

$A \rightarrow B \rightarrow C \rightarrow D$

$A \rightarrow E \rightarrow D$

If a name clashes problem exists i.e.  $m1()$  is redefined in C, B or E, D inherits only one version of  $m1()$ : either the method explicitly refers to the desired definition i.e. originator parent class, or a default scheme is provided by the support language. Thus, one of the two paths has to be dismissed for the study of  $m1()$ . When the call to  $m1()$  is explicit, the reference to the originator class is given (see section 2.1.8). When the call to  $m1()$  relies on the default scheme provided by the programming language, the default path is then the chosen one.

While detecting the various inheritance paths is straightforward, assessing if the methods in D are redefined necessitates an investigation of the code of methods in D to detect which versions are explicitly referred to. Otherwise, if the designer relied on the default inheritance scheme to obtain the desired functionality and to remove the ambiguity, a metric's collector will have to implement the corresponding algorithm. Technically, a possible solution to discover multiple paths relies on the parsing of the concerned classes for extracting the associated parent and child classes. However, in languages that provide reflective capabilities such as Smalltalk [GolRob90], parsing is not necessary as appropriate functionalities permit the discovery of inheritance relationships between classes.

### Class wrapper

This section explains a technique based on *wrappers* to filter out desired information from an OO design. A class wrapper would aim at analysing class internals and intercepting its interactions with other classes. In general, wrappers are used between two applications for intercepting the set of transiting messages. For example, the *tcp\_wrappers* [CheBel94] are a set of API functions that shadow the real functions based on tcp communications e.g. telnet, ping, finger, etc. When a client program initiates one of the cited functions, a corresponding *tcp\_wrappers*<sup>15</sup> function takes

<sup>15</sup> Note that the concept of *proxies* for web servers provide similar functionalities.



control, filters out the wanted information and launches the real invoked function. In such a way, the execution of the wrapper function is completely transparent, does not interfere with the execution of the real function and dynamically extracts the wanted information. A class wrapper acts in the same way for static information. The wrapper encapsulates a class in order to extract meta information such as the class definition and the details of interactions between classes such as method sender, message sent and method receiver. Note that meta class information such as messages sent or received, number of parent and child classes or number of methods are possible candidate metrics themselves [LorKid94].

In Figure 3.11, the design of a possible class wrapper is shown. It includes two parts acting as the filters for the desired information. Indeed, the filters are configurable in the sense that only the wanted information would be filtered out and addition or removal of other filters is possible. ClassC is scrutinised for extracting information such as the list of instance or class variables and methods, the list of ancestors classes, the list of external methods internally referenced and the list of external methods which reference internal methods.

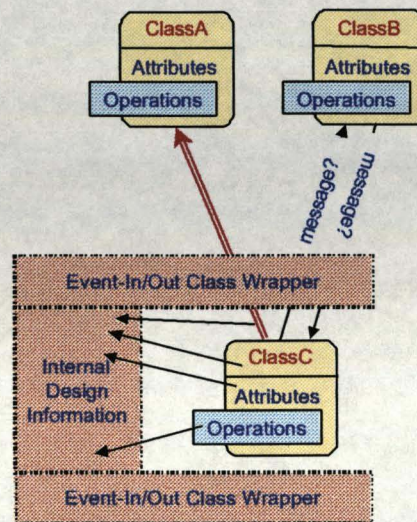


Figure 3.11: Class wrapper

A class wrapper may exist under the form of a set of API functions, therefore they could be closely integrated with a metrics collection tool. Collaboration for information exchange can take place between the client metrics tool and the wrapper functions.

### Hierarchy wrapper

In Figure 3.12, the technique of class wrappers is extended to a branch of a class hierarchy. Particularly for the assessment of behavioural inheritance, it is interesting to isolate a branch of the hierarchy for a detailed study. A hierarchy wrapper would mainly rely on information provided by the class wrapper at a lower level; however, the filters would provide information on all classes of

the branch instead of a single one at a time. In such a way, comparison and use of the design information are made easier either for design analysis or for deriving metrics on the hierarchy.

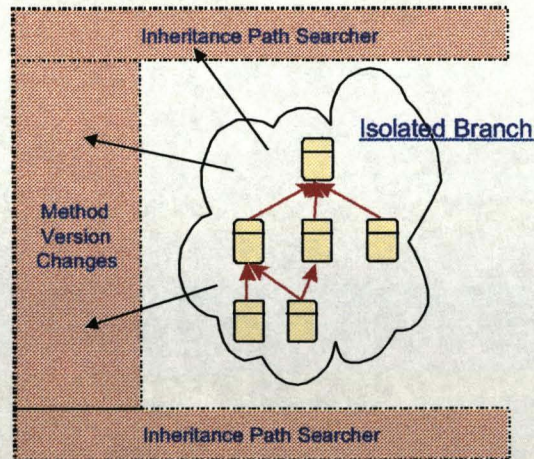


Figure 3.12: Hierarchy wrapper

It may be useful to build a design repository which would be persistent. Such a repository would include all characteristics for each class of the branch together with their relationships with other classes. This implies that the hierarchy wrapper can be invoked independently of a client program. Alternatively, like for the class wrapper, the hierarchy wrapper would be closely integrated with the collection tool for dynamically extracting information “on the fly”. The benefits of having hierarchy level information as opposed to class level information is that the analysis of inheritance paths is possible, therefore the history of method changes down a branch can be followed. Also, due to the consequences of property modifiers, the detection of cancellation of methods permits the suggestion of potential wrong subclasses.

To date, the concept of wrappers has not been applied in the context of a measurement programme. For the purpose of assessing class hierarchies, the technique is convenient and permits an encapsulation of the two levels: class or (branch of) hierarchy.

The next section concentrates on the different method states when being redefined. A state transition diagram is used to illustrate the relevant transitions.

### 3.3.4. State transition diagram for the method redefinition mechanism

The assessment of the mechanism of redefinition requires a deeper analysis of the methods present in the hierarchy. The tracking of the evolution of method status becomes essential from an assessment perspective. This section introduces a state transition diagram that captures the possible states of a method when being redefined down the hierarchy. A set of expected and unexpected transitions is explained.

In most OO methods literature, the mechanism of inheritance is illustrated in examples involving a parent and a child class. Although the case of multiple inheritance involves many parents, the coupling effect is still shown for the pair of parent-child classes. Managing many levels of depth requires an overview of the whole hierarchy or at least a separate view of the branches. Due to the transitivity of the inheritance relationship, for each of the inheritance paths, publicly declared properties are passed from one level to the next level of depth down to the leaf class. For this reason, correctly extending an existing hierarchy requires a good knowledge of the design of ancestor classes. This adds an additional burden for the designer in the case of off-the-shelf class hierarchies. Three main factors affect the designer's choices when looking for appropriate abstractions in existing hierarchies:

- **Class complexity vs. depth:** the behaviour of classes increases in complexity when many levels of depth are involved. In the case of commercial class hierarchies, the decision for extending the hierarchy is often based on a limited number of factors due to the size of the hierarchy and the number of possible dependencies. The consequence is that the chance for wrongly extending inheritance is higher.
- **Accumulated inherited properties:** the size of accumulated inherited properties may become un-manageable by designers if the classes encompass a large number of methods. This directly affects the decision for the solution design and often induces ignored inheritance in the hierarchy.
- **Class and behaviour documentation:** the availability of a comprehensive description of the classes and behaviour is always desirable but not present in many cases, thereby making the reuse of the classes difficult. The existence of examples is a crucial factor to the understanding of the existing classes and associated methods. The *Javadoc*<sup>TM</sup> software tool from Sun Microsystems<sup>TM</sup> directly addresses this point. Given that pre-defined tags have been inserted in the Java source code, *Javadoc* formats the public API into a set of HTML documents, thereby providing the detailed description of classes and methods in a standardised way.

For a class, for each attribute and method, the scope modifiers define the encapsulation of the class, thus future heir class visibility. Inherited properties mean that they have been declared as either public or protected in the parent class. In such cases, various changes can be done to the implementation of an inherited method. In order to visualise the effect of change of state of methods from a parent class to a child class, a state chart diagram is used in Figure 3.13 (see

section 2.2.4 for a description of redefinition variants). From an assessment perspective, the change of state of methods from one level to the next level permits accurately following their evolution in the branches of the hierarchy. The method's state refers to the changes happening to an existing method between its version in a parent class to the version in one of its child classes.

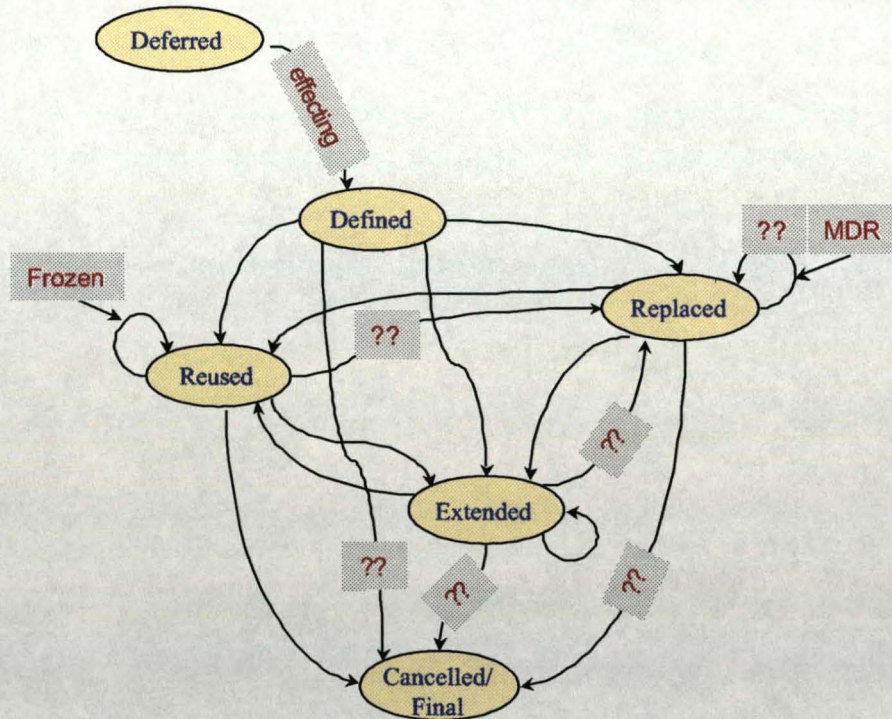


Figure 3.13: State-chart diagram for method redefinition<sup>16</sup>

In Figure 3.13 all possible state transitions of a method are represented. Six different states are listed:

- **Deferred:** when a method is in a deferred state, the only next possible state is being defined. Eiffel refers to the action of providing the first method definition i.e. body, as *effecting* the method. This is also known as *realising* the method.
- **Defined:** it is the first time definition for the method.
- **Reused:** the method is reused without modification.
- **Extended:** the inherited implementation of the method is reused with addition of new code.
- **Replaced:** the method is completely replaced, the signature remains the same.
- **Cancelled:** the method is removed from the child class
- **Final:** the method is declared as non-modifiable although accessible.

<sup>16</sup> Note that the two states "Cancelled" and "Final" are treated as a single state as they both are questionable states.






 Parent to Child	Defined	Reused	Extended	Replaced	Cancelled /Final
Deferred	effecting				
Defined					
Reused					
Extended					
Replaced					

Table 3.5: State transition table for method redefinition

Key	
	Not applicable. From the <i>deferred</i> state, method can only become <i>defined</i> .
	These transitions represent the recommended use of inheritance <sup>17</sup> .
	Questionable transitions mainly occur when the final state is either <i>replaced</i> or <i>cancelled</i> .
	MDR problem. See section 3.1.2 for a detailed description of the multiple descendant redefinition problem.

The main purpose of the state transition diagram is to detect suspect or unexpected method changes down the branch of the hierarchy. During the extension of a class hierarchy, different design constraints may appear whether an existing class library is provided and reused as it is. If so, the process of investigation of the wanted abstractions (i.e. (set of) classes) constitutes an important task in the design process. Pragmatically, designers or implementers rely on a localisation of an appropriate branch of the hierarchy in order to reduce the search range. In current class hierarchies, abstractions are fairly well-decomposed and organised as branches of the hierarchy. For example, current graphical interface abstractions also referred to as *frameworks* are well established and solve most of the needs of information systems requirements.

In this document, the focus is given to transitions (Table 3.5) which might suggest design problems. Mainly, it concerns methods whose state is either replaced or cancelled. Although Meyer [Mey97] promotes method overriding under the condition that the semantics remain the same, the checking of consistency of the semantics is difficult. In detecting the change of state of methods down the hierarchy, there are opportunities to suggest potential inconsistencies in the use of the redefinition mechanism.

Clearly, the detection of suspect state transitions is desired; however, it should be noted that further complex method redefinitions that are not captured by the state transition diagram presented above could take place. Such complex redefinition cases, often obscure, are presented in the next section.

<sup>17</sup> Eiffel provides a construct which fixes and disallows future changes to a method. Such a method is referred to as *frozen* and is equivalent to a *final* method in Java. Note that for the transition: *Reused* to *Reused*, a frozen or final method can be reused.

### 3.3.4.1. Remark: method redefinition and unexpected message sends

The state transitions for methods described above aim at suggesting potential wrong use of method redefinition; however, further complications may occur. From an assessment perspective, it is essential to be aware of such situations that cannot be easily detected automatically. Although a redefined method may seem conceptually valid, developers are offered many opportunities to deviate from the inheritance scheme when implementing the method. Sometimes the context may require a portion of code qualified as a *hack* to provide a simple solution to a problem e.g. in the case of inheritance of somebody else's code. However, a dangerous situation may happen in the case of careless programming. A redefined method may appear correct from the point of view of its interface but not from the point of view of its semantics, therefore incurring consequences on previously made assumptions on the design.

The possible combination of message-passing, the delegation mechanism and the effect of encapsulation are the main causes of the problem of unexpected messages in the method's implementation. Message-passing generates dependencies between objects but also affects the validity of inheritance because of method invocations in non-conventional ways. Such method invocations results from the hazardous use of directed resends i.e. ability for an overriding method to invoke the overridden version (Smalltalk-80 has `super`, CLOS has `call-next-method`, C++ has qualified messages using the `::` operator [Cha97, Ste90, Str90]). In Figure 3.14, four classes a, b, c and d with  $d < c$ ,  $c < b$  and  $b < a$  are represented.

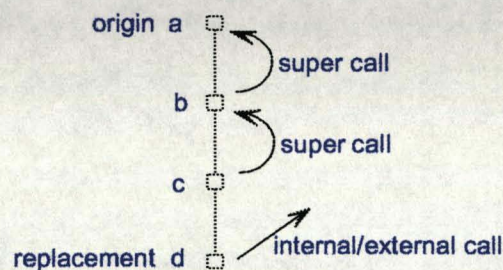


Figure 3.14: Expected method invocation

Consider class  $a = \{<m()>\}$ , with  $m$  defined as public. Method  $m_b()$  and  $m_c()$  are extended methods, therefore an invocation of the ancestor implementation is made via the `super` call. Method  $m$  is replaced in class  $d$  therefore its implementation is completely different from its parent one and it is expected that the semantics would remain the same. Note that method  $m_d()$  is entitled to send messages to other remote methods i.e. internal or external calls. In a redefined method, three types of invocation are possible: reference to the closest inherited parent's implementation, explicit reference to an inherited parent's implementation<sup>18</sup> and other internal or external references

<sup>18</sup> Note that the difference between a classic `super` call and an explicit `super` call is that, in the latter case, the ancestor's identifier is specified in the call, allowing the caller to refer to a specific parent's implementation of the inherited method.

to the class. The combination of those possible references gives opportunities to deviate from the correctness of the inheritance.

In Figure 3.15, four examples of such cases are given.

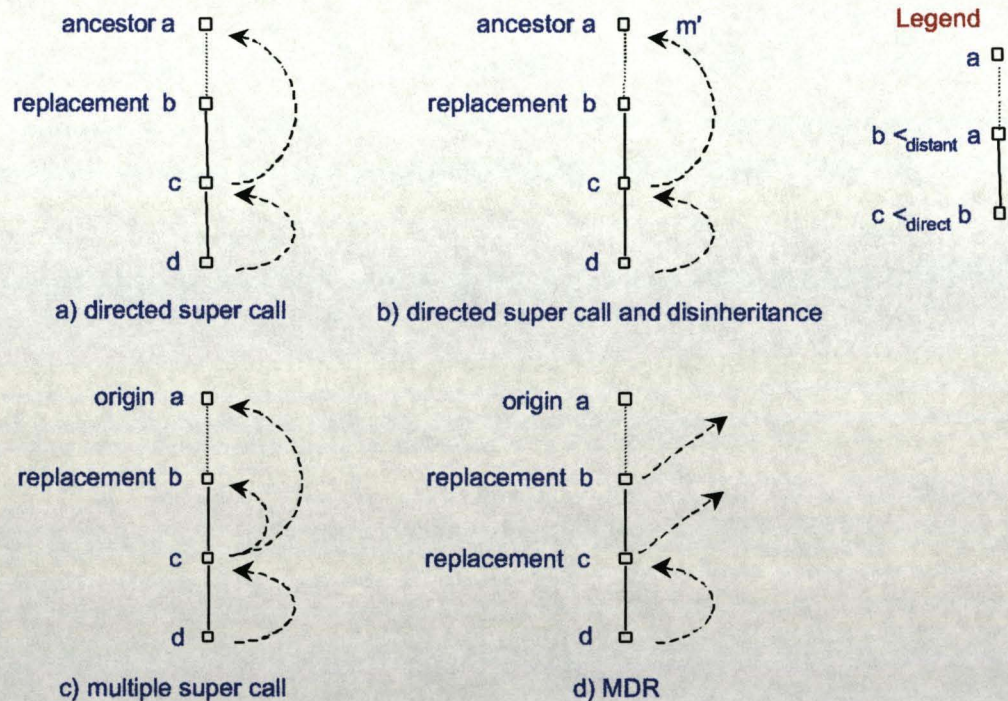


Figure 3.15: Examples of unexpected method invocations

Examples in Figure 3.15 show various uses of method replacement. In case a),  $m_d()$  and  $m_c()$  extend their definitions whereas  $m_b()$  replaces the method.  $m_c()$  issues an explicit **super call**, not to the latest inherited implementation (from class b) but to one of its previous ancestor (class a). Effectively, a previous implementation of method  $m$  is wanted for the class c. Thus,  $m_a()$ 's implementation is not available for reuse unless referenced within an explicit **super call**. Note that either or both class b or c are considered as suspect classes i.e. abnormal case of inheritance.

The case b) is a variant of case a).  $m_c()$  issues an explicit **super call** to a method different than the (inherited and redefined) method  $m_a()$  i.e. **super call** to  $m'_a()$  and  $m'_a()$  publicly defined in the superclass a.  $m_c()$  completely changed its original semantics and in addition, it refers to a different method in one of its superclasses which suggests that  $m_a()$  and  $m'_a()$  may be variants of each other. This clearly suggests a design problem as the semantics are different than the original.

In case c), an example of multiple **super calls** is given.  $m_b()$  replaces the inherited implementation therefore no **super call** appears. In order to extend the inherited implementation,  $m_c()$  issues two **super calls**: one as normal and one to the previous ancestor's implementation for code reuse. As  $m_c()$  reuses  $m_a()$  and  $m_b()$  implementation, this seems to be a possible way to simulate multiple

inheritance although not a satisfying design. Class *c* re-establishes the expected inheritance scheme. Class *b* is a suspect class.

In case d), an example of MDR is shown. The multiple method replacement implies that  $m_a()$ ,  $m_b()$  and  $m_c()$  are different versions of the same inherited method. Further subclasses to *c* must use the directed super call mechanism to reactivate “lost” implementation resulting from previous method replacement. Given the definition of the MDR in section 3.1.2, the referred parent-child relationship between *b* and *c* (Figure 3.15 b)) is a direct relationship. However, other unexpected situations related to the MDR problem may appear and are described below.

Consider Figure 3.16 where two scenarios, referred to as *distant MDR*, are shown.

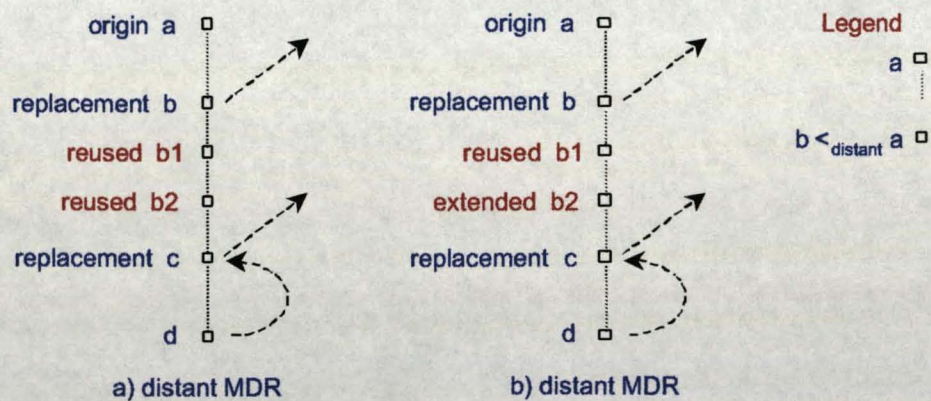


Figure 3.16: Distant MDR scenarios

Rather than a direct inheritance relationship between the classes *b* and *c*, they may be separated by other classes *b1* and *b2*. Whereas *b1* and *b2* are only reusing  $m_b()$  (Figure 3.16 a)) or reusing and extending  $m_b()$  (Figure 3.16 b)), a subsequent replacement i.e.  $m_c()$ , raises further design issues. Intuitively, such complex sequence of calls does not suggest any recognised appropriate use of inheritance and is not well understood. Such situations may be attributed to optimisation reasons in class hierarchies i.e. only the behaviour in the leaf classes is completely re-implemented for performance. Often, these classes are also defined as finalised (see 2.1.7).

The examples of unexpected calls described above demonstrate that designing classes using only method interfaces does not ensure a correct design. This contradicts the claim of current methodologies for completely decoupling design issues from implementation. The use of inheritance and the design of method interfaces rely on assumptions on the inheritance scheme, which may not hold at implementation phase. More importantly, such situations affect the maintenance of the application but also distorts metric results as they may be categorised as correct measures. To prevent hidden method redefinition abnormalities, code inspection is desired. The state transition diagram described in 3.3.4 cannot detect such anomalies either. Currently, only an analysis of the source code permits the detection of such problems. Alternatively, in a dynamic event model (see OMT methodology [Rum91]), as the message flow is defined, it reduces the



chance for the problem of unexpected calls. Nevertheless, the design of a class hierarchy with the intention of code sharing and code reuse is not only an application-solving problem but also a software engineering activity.

The next section provides a synthesis of the behavioural inheritance analysis technique which is aimed at providing a visual representation of the method's life history in class hierarchies.

### 3.3.5. Behavioural inheritance analysis

Providing that an object model is stable i.e. towards the end of the design phase, it is essential to gather an overview of the architecture and design issues involved for the entire application. This may be seen as a design validation phase or a final design review phase preceding the implementation phase. One important aspect for the design of class hierarchies is to make sure that the semantics of the behaviour are correct for optimising reuse regarding the inheritance use and the set of requirements for the application. Behavioural inheritance analysis addresses the problem from the interface point of view. Three techniques have been described in sections 3.3.2, 3.3.3 and 3.3.4:

- Obtaining design information useful for metrics.
- Class analysers and inheritance path isolation.
- State transition diagram for method redefinition.

Using the output of each of the above techniques, the aim is to build a snapshot of the life history of methods in a particular branch of the hierarchy. For each of the classes of each path of a hierarchy, the method is analysed to record the evolution of its state. A possible representation is to reproduce an image of the concerned hierarchy with addition of method's state to provide an overview of the method's life history.

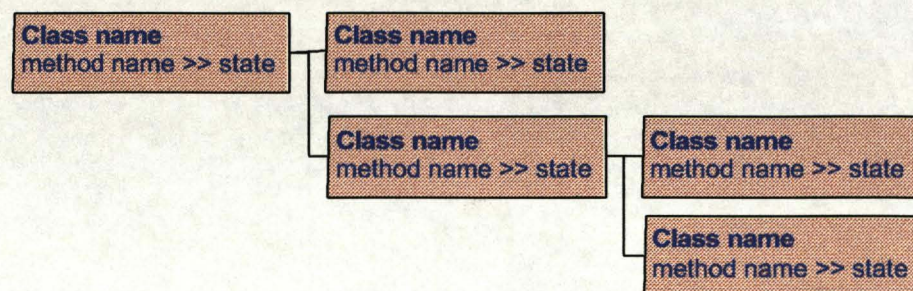


Figure 3.17: Method life history representation

In Figure 3.17, an exact reproduction of the concerned branch of hierarchy is used to show additional information about the methods. Thus, the top class is the main parent class for all isolated paths found. Recall that the idea for studying a branch is interesting because it captures a set of related concepts such as the Collection branch, the Stream branch or the WindowManager

branch. For each method defined in the main parent class, all the paths are scrutinised to show how the method evolves further down the hierarchy. For a particular branch of the hierarchy, the method states are recorded in each class in the following form:

method name >> state

with state = {deferred, defined, replaced, extended, cancelled} (see section 3.3.4)

A behavioural analysis may only concern a subset of methods which have to be evaluated, therefore, not all existing methods in the class may be displayed. Recall that methods simply reused in a class except for the case of super calls are not shown as it requires a detailed parsing of the code for detecting such cases. Therefore if a class does not show a method, it does not mean that it is not used. Various possible ways of use or reuse include aggregation, inheritance, message passing or arguments of methods that cannot be detected by previously described techniques. Thus showing a limitation of the behavioural analysis technique.

An example of use the technique is illustrated in the next section.

### 3.3.5.1. Experiments on the Collection class

In Figure 3.18, the example given in section 3.1.2 is revisited. Only the relevant methods from the Collection class are shown. To obtain the hierarchy below, the designer will rely on a tool and therefore additional filtering mechanisms to organise the information e.g. one method at a time, only replaced methods or a specific isolated path, are possible.

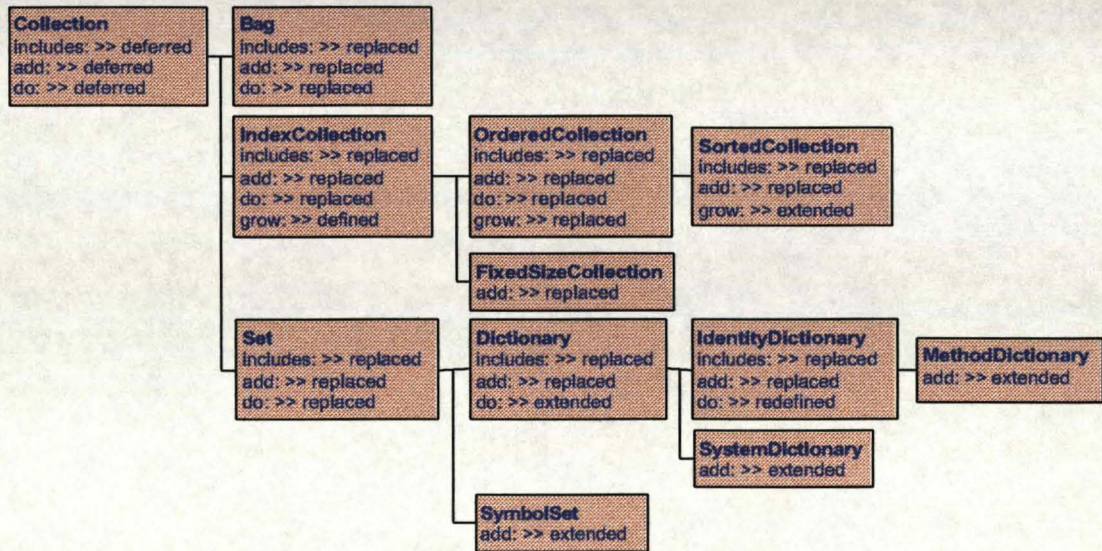


Figure 3.18: Method life history for the Collection branch

The representation of the class hierarchy together with the method s status permits an overall view of the life history of the method. This representation clearly pinpoints the problem of MDR in the includes:Collection and add:Collection methods. Although originally declared as abstract methods, in

most paths the methods are completely redefined. The few exceptions concern the `add:` method in the classes `MethodDictionary`, `SystemDictionary` and `SymbolSet`. Particularly, the path `Collection`, `SetDictionary`, `IdentityDictionary` and `MethodDictionary` raises attention. All versions of the `add:` method have been replaced except that in the leaf class where the method is extended. Having a case of MDR and an extended method in the leaf class suggests that inheritance is used for sharing of other methods not shown here. Although all classes in the same path seem to be structurally similar, the semantics seem to be different according the evolution of the `add:` method.

Another interesting case concerns the `grow:` and `do:` methods. The `grow:` method is firstly defined in the `IndexedCollection` class. Then, subsequently replaced in `OrderedCollection` and finally extended in `SortedCollection`. Besides the fact that the method has been replaced once, not enough arguments allow us to conclude that it might be a problem. On the contrary, the fact that the method is extended in the leaf class gives it credit. The case of the `do:` method is the opposite. After being replaced, then extended, it is again replaced. This raises a “design alarm” for potential incorrect interface design. Notice that if a method is declared as deferred in a parent class, the first replacement is a correct use of the redefinition mechanism (see section 3.3.4, Table 3.5 for the recommended transitions).

Design decisions are not possible at this stage, as other methods in the classes should be considered. This is the reason why measurement techniques will complement such analysis. For example, given a ratio of replaced methods compared to the number of extended methods gives an indication on how the redefinition facility is used in the model.

Following the description of all design considerations relevant to the assessment of inheritance in the above sections, the interpretation and understanding of use of the inheritance relationship and the method redefinition were clarified. This reduces the chances for ambiguities, and thus enables the delivery of an appropriate measurement programme.

In a measurement process, not only the definition and derivation of the metrics are important. In the literature on measurement, the topic of interpretation process is little described. Paradoxically, it is generally agreed that without a good design feedback from the analysis of the metrics results, a measurement programme may fail. The next section proposes a novel data interpretation framework for the assessment of OO models. Emphasis is given to the necessity for generating sensible feedback to the designers. In addition, the data interpretation framework aimed at being integrated as the final process within the GQM model.

### 3.4. Mechanisms for data interpretation of metrics for object-oriented systems

*"There are as many scientific methods as there are individual scientists"*

– Percy W. Bridgman, On "Scientific Method"

#### 3.4.1. Introduction

Measurement techniques are valuable and troublesome design tools at the same time [Avo94, Boo89, BinSch96, Bri96, ChiKem91, KosVih92, McKMon93]. The analysis and interpretation method used is an important component in the measurement process. More than a simple data retrieval and representation mechanism, the analysis and interpretation technique should be designed to illustrate a few particular aspects of the feature assessed [Ebe92]. Because of the relative immaturity of the OO metrics research field, little research has been done on the interpretation and analysis of metric results, making meaningful design decisions difficult. For example, in the *depth of inheritance* (DIT) metric, Chidamber and Kemerer interpreted the possible results as, "top heavy" (too many classes near the root) or "bottom heavy" (many classes near the bottom of the hierarchy) designs. However, whether a class hierarchy falls under one or the other case seems arbitrary, and thus subjective.

Measures are only significant if they are objective and repeatable. Metrics that require subjective assessment where a range of complexity values are arbitrarily affected have been recognised to have no scientific validity [Hen96]. Complexity values may be used for attributing weightings to the metrics. Instead, it is preferable not to take into account subjectivity that makes the data interpretation difficult. Stating that a design is good is only valid with respect to particular criteria. One such criterion might be the non-dependency of classes to other classes, which exhibits a low level of class coupling. In addition, the qualifier "low-level" must be related to a hypothetical average or threshold for the particular metric under consideration. Interpretation of data relates to the goals and assumptions stated for the concerned metric. For example, an assumption concerning the DIT metric is that the deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, therefore the more complex it is likely to be. So, a typical DIT curve would decrease rapidly on a *number of classes per DIT* graph. Currently, the metric results analysis is carried out in a pragmatic way. Outstanding patterns or phenomena drive the process. Often, a graphical display provides assistance for quick and easy feedback over a table of numbers.

Although the area of representation and visualisation constitute separate topics of research, they strongly relate to the interpretation techniques used in a measurement programme. To date, emphasis has been given to the early stages of the measurement programme. A consequence of this is that metrics have been criticised for collecting large amounts of data without any suitable methods for analysing the data afterwards, making them useless [Fen90].

Currently, the interpretation of metric results rely on one possible understanding of the OO concepts, which is the reason why the emphasis is put on the early *goal definition* stage for a candidate metric. In general, most authors use statistical methods in addition to empirical analysis methods but others have emphasised the need for more appropriate techniques [Abb&al94, Bak&al90, Bar&al93, Bas&al94, Bou89, Bri96, BriCuc98, Hen96, RosHya96]. Clearly, the difficulty of interpreting metric results asks for complementary analysis techniques. This section argues that, depending on the characteristic assessed, the combination of a dedicated analysis and interpretation technique and the use of appropriate graphical representations procure additional and better quality information feedback from the metric results. In addition, other supporting tools for pre-processing and data analysis may be required. For example, trigger rules that characterise a particular phenomenon on a given curve can be defined and automatically detected. As the “goodness” in a design is subject to disagreement because it depends on the interpretation of each, appropriate analysis and interpretation techniques must take into account the variety of characteristics assessed, the environment and the purposes of measurement. Thus, the efficiency and relevance of metrics relates to the amount of feedback produced about the design and the suggested ways for improvement.

The methodological issues involved in the interpretation process are defined as follows:

- A description of examples of convenient data visualisations for a collection of metric results. The benefits and drawbacks of each are highlighted.
- An exploration of possible utilisations of pattern profiles with regard to the intrinsic properties of the data visualisation type.
- A novel interpretation framework is proposed. The detection of particular design problems is realised using an “*alarmer*” technique and triggered conditions.

The following sections propose a data interpretation method based on pertinent visualisation of a data set obtained from the method redefinition metrics for object-oriented systems. The data interpretation method aims at facilitating the metric results interpretation, the design problem identification and constitutes a means of deducing design decisions. It is discussed how this method constitutes a solution to re-injecting design information in an object-oriented model. This work aimed at the generalisation and integration of the data interpretation method within a design evaluation cycle framework.

#### 3.4.2. Motivation and approach for interpretation

In the current literature on assessment methods for OO systems, the importance of extracting design information feedback from metric results [Hen96, LorKid94, Whi97] has been highlighted. However, to date, emphasis is still given to the correct definition of metrics and the goals for which they are defined. Then, the data obtained from derivation of metrics are empirically studied.

The data visualisation method presented in this section is based on the idea of *metric profiles*. Any deviation from this “norm” will suggest potential inconsistencies which correspond to specific design problems. However, the deduction of conclusions from raw data obtained from metrics is not straightforward. One way to tackle such problem is to provide a complementary method or technique for designers to facilitate the measurement process.

Three main aspects are considered in our analysis and interpretation method as follows:

1. In general, raw data are pre-processed before being analysed. The nature of the processing function is chosen depending on the type of results expected. For instance, only a range of the values may be relevant at a time, or the values may be more suitable for reading on a logarithmic scale. Any transformation of the raw data contributes to the overall method for analysis.
2. The use of graphical representations directly depends on the type of values returned by the metrics and the purpose of measurement. Based on the assumption that different visual representations are able to express different aspects of a measure, considerations have been given to the investigation of a set of representations applied to the same set of results. Such experiments enable the interpretation of the metric results from different angles.
3. The need for additional interpretation aid tools such as searching or querying facilities also contributes to the interpretation process. When the graphical representation includes a large data set, details are not necessarily obvious to the human eye. To un-clutter the graphic with unwanted data, several techniques can be used e.g. zooms, filters, triggers, data transformation. Identified and recognisable patterns for a profile can then be detected automatically e.g. increase of rate by a factor of  $x$ . However, from an investigation point of view, the designer may not know in advance what to expect concerning the characteristics of the metric profile. In such a case, it is likely that the needs for appropriate tools are only identified during the interpretation process. Such methods, similar to a “data mining” activity, are usually dedicated to a specific purpose contributing to the interpretation of the behaviour observed.

Interpretation techniques are highly dependent on the properties of the attributes assessed. The interpretation stage is only part of the measurement process, it is nevertheless, crucial for the delivery of the expected benefits. Recall that the outcome of a measurement program can be either:

- **Expected.** In such a case, it means that the result obtained is expected to match the predicted result. Providing that the notion of “goodness” or “badness” is defined, the difference between the values gives indications on the quality level of the attribute. Expected results permit the confirmation of general hypothesis such as:

*“due to the abstraction level of classes situated near the top of a class hierarchy, the deeper a class is, the higher the level of redefinition”*

*“for a DIT level, a high level of redefinition may suggest a potentially design problem in the current level and parent levels affecting the understanding, maintainability and extendibility of the class hierarchy”*

or more specific ones such as:

*“a redefinition level higher than 50% indicates a potential MDR problem arising at the considered DIT in the hierarchy”*

*“the ratio of extended methods compared to the total redefined methods gives evidence of a class reusability”*

*“a method redefinition rate increase > 30% suggests the presence of the MDR problem”*

Note that the above mentioned thresholds may be based on existing benchmarks.

- **Unexpected.** In such a case, the interpretation is open to suggestions arising from the observation of the metric results obtained. An empirical study the profile obtained ought to discover particular patterns for further investigations.

Whether the metric results are expected or not, the desired feedback provides explanations or suggestions for improvement concerning the observed profile.

In section 3.4.3, a novel interpretation framework is presented and used for the evaluation of different types of graphical representation. The framework addresses the lack of the GQM approach for the analysis of the metrics results.

### 3.4.3. Metrics interpretation framework

*“The capability to qualify a process or product with measurement data is limited by the abilities of the analysts.” – Henderson-Sellers*

Goodness and badness are two possible quality design attributes. Inevitably, a design always shows weaknesses regarding some particular OO aspects while presenting strengths in other aspects. The area of measurement contributes to the design decision process and helps in the identification of recognisable design anomalies. Often, comparison is adopted as the technique for interpreting metrics results. However, as stated in [Ban97], the designer should make sure that the metrics values are comparable at first. To compare an aspect to another, they must be related to each other i.e. variants or serving the same purpose. In addition, they must be in related context e.g. similar conditions for comparison. In this thesis, the aim is to assess the various uses of the redefinition techniques. So, the measures are compared to each other within the same branch of the hierarchy. When two different branches belong to two different categories (see interpretation given to systems in sections 5.4 and 5.5), comparison is only made from a general perspective of use of

redefinition and conclusions can be drawn regarding the different type of profiles obtained. The metrics interpretation framework proposed in this section ought to minimise the risk of incomparable data in guiding the designer for the choice of the correct representations and interpretation mechanisms.

The proposed metrics interpretation framework is aimed at being integrated in a traditional measurement process such as the GQM model. In consequence, the given description assumes that the interpretation phase naturally takes place after the *metrics collection* phase. In the light of explaining the crucial stage of data interpretation, it is necessary to re-visit the design and measurement process to demonstrate the strong dependency relationship involved between the early stages of design and the final stage of a measurement programme. In a software development process, the designer's perception is the core element to the success of the realisation of applications.

In the following sections, the importance of the designer's perception is highlighted. It is shown how the interpretation framework can be decomposed in the three following aspects: raw data representation, profile analysis and design feedback. Details of interactions between different components of the framework are explained.

#### 3.4.3.1. Designers' perceptions and decisions

An interpretation process is a reasoning activity. As the decision making process is done by the designer, many factors influence the final decision. The designer's experience is one such factor (Figure 3.19). If an empirical analysis approach is adopted, the interpretation starts with an observation phase where an overview of the data is analysed. Then, a more detailed study is necessary. It is noticeable how the designer's perception or understanding of the underlying OO concepts affects the conclusions of an interpretation process. For this reason, the knowledge of the intention of the designer when the candidate design was built is crucial to the interpretation phase. External subjective factors may also compromise the interpretation as well validating it. For instance, experiments illustrated in [Abb&al94, Ban97] proposed to choose evaluators i.e. designers based on similar experiences to rate a set of aspects of design. The results of the experiments showed a general consensus on the quality attributed to each design. However, it can be argued that in such a situation, there exists a degree of subjectivity related to the quantification of the level and similarity of experience of the designers. The number of years may be one possible approach to quantify such level. In consequence, in an interpretation process, the less subjective it is, the better the quality of the conclusion is.

An interpretation process is also based on the understanding of the OO concepts used. It is therefore important to relate the designer's perception of a concept with the interpretation of a measure. This is particularly important in the case of use of an OO principle that exhibits different interpretations itself.



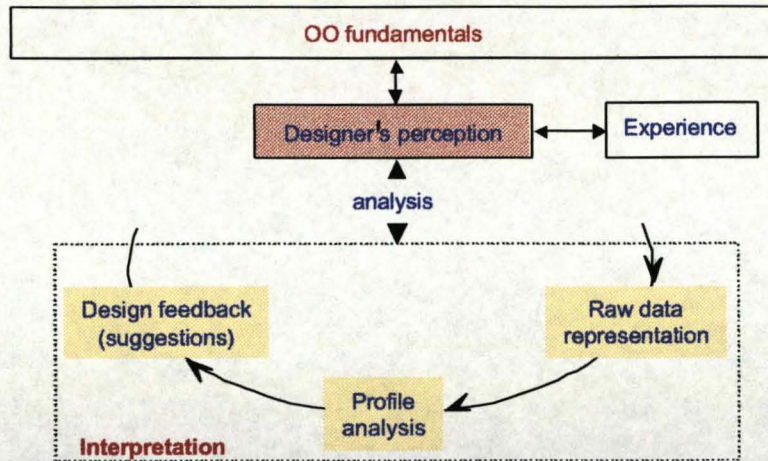


Figure 3.19: Analysis, interpretation and interactions

The interpretation process can be decomposed into three aspects (Figure 3.19):

- The representation of the raw data set implies that the metric results are not processed before display.
- The analysis of the profile represents the process by which extraction of the design feedback is possible.
- Design feedback. Often, this involves a comparison of the metric values obtained against the assumptions made on the OO characteristic assessed.

After a presentation of the benefits of graphical data representations, a detailed description of the profile analysis task is given in the following sections. In particular, the interpretation techniques focus on the discovery of unknown design features.

#### 3.4.3.2. Raw data representation

To date, most research work on metrics has concentrated on the metrics themselves and does not exploit the results from different perspectives. The derivation of metrics tends to generate a large data set as a result. Therefore, a graphical representation of raw data is the first natural step. Instead of a table of plain numbers which might be suitable in some cases, the main benefits of a visualisation is that it is easy to pinpoint disparities. The evaluation of different representations is desired in order to identify the appropriateness of these with regard to the metric chosen and the design characteristics expected to be interpreted. The suitability of the visualisation type chosen determines the correctness of the interpretation.

The data representation phase is illustrated by the three components shown in Figure 3.20.

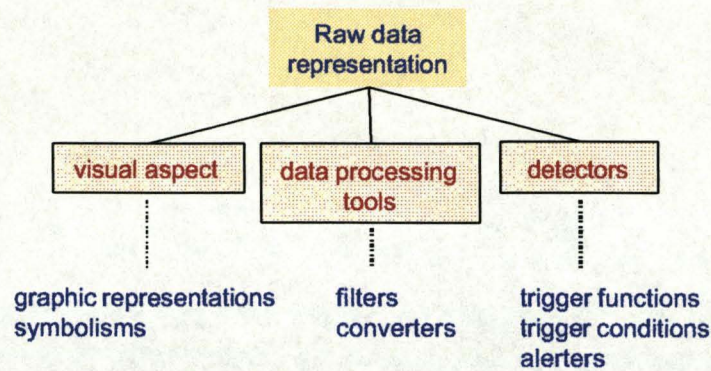


Figure 3.20: Data representation

### Visual aspect

An advantage of use of graphical representations is that they are not limited to well-defined ones such as bar charts, surface charts, etc. Several types of diagram may be appropriate for the same data sets offering the choice of many perspectives. For example, symbolic diagrams [Ebe92] can reduce information content while increasing readability and clarity. The symbols are arbitrarily chosen according to the values.

The motivations behind the use of graphical representations for the method redefinition profile are manifold:

- Ease of analysis, comparison and interpretation: a visual representation is, in most cases, more convenient than raw data sets, especially large, in a table. The type of representation or symbolism used determines the expressiveness of the visual aspects. For instance, in the case of a ratio values type in a data set, the pie chart is one possible representation.
- The comparison of the redefinition activity for different branches is made easier. A redefinition profile can act as an element of reference in a comparison. The investigation of differences between two shapes indicates similarities or dissimilarities of the design from the point of view of behavioural inheritance.

### Data processing tools :

Sometimes it is convenient to transform raw data (Table 3.6) before it is visualised. Examples of use of data processing tools can be the extraction of a reduced set of data, the data transformation into a different scale unit or the conversion of the data into ranges for enabling different perspectives. A pre-transformation of data is seen as a re-processing stage where the results from the transformation are expected to exhibit some desired features or peculiarities. The possibility of hiding or showing a subset of the raw data set is crucial for the analysis and interpretation of the metric's results. The focus on certain aspects of large data sets permits the discovery of details

which are otherwise unnoticeable. Although rare, data sets may also contain redundant values that can be removed by a filtering function. As applications evolve in time, it is also possible to see the effect of changes made between two versions of the same class, branch or system in comparing different versions of the redefinition profiles. Such comparisons are made easier with the presence of graphical representations.

DIT	Redefinition profile (%)
1	6.48
2	19.39
3	42.15
4	56.03
5	45.54
6	52
7	60

Table 3.6: Smalltalk Express Object branch redefinition profile

The choice of the *transformer function* is outside the scope of this thesis; however, transformations in the metrics domain are considered.

### Detectors

In general, abnormal or unusual values indicate abnormal or unusual design features. The discovery of such unusual values may be straightforward if visual. As the redefinition metrics are mainly utilised to assess branches of hierarchies, depending on the size of the branch, a fine detection of potential suspect classes can be done due to the derivation level by level. The technique of detectors is complementary to data processing tools as the latter can be used as a filtering system to reduce the amount of data processed. A data interpretation model using alarmers is presented in section 5.11.

Providing that suitable visualisation of the metric results exists, one possible way to identify design inconsistencies, for a given characteristic, is to assess the disparities on the graphical representations. This leads to the notion of *pattern profiles*. An example of detectors used in the experiments is the technique of *alarmers* (section 5.10) which are aimed at specifying and recognising such disparities. More generally, the identification of conditions under which a disparity occurs is essential for design problem detection.

Ideally, it is sought to recognise typical pattern profiles which would be classified for a particular metric and thereby, the corresponding design problems. Suggestions for design improvement would then be facilitated. A profile should exhibit some expected characteristics or properties related to the metric considered. An alternative choice is to look at the range of possible chart types available for evaluating their appropriateness against the concerned metric. Not all graphic representations are suitable for a given metric, the choice depends on its type, on its properties, on the characteristics to be measured and on the type of results expected. For example, the

redefinition metric set measures the amount of redefined methods in a class hierarchy. The measure is taken level by level in the hierarchy and a percentage is returned for each level. Therefore, the type of results is discrete which prohibits the use of smooth curves. Instead, visual representations such as bar charts or scatter plots are the most suitable. Novel visual representations and symbolism are encouraged for the representation of results, especially if the properties of a particular phenomenon are known i.e. conditions under which a phenomenon is likely to appear. Although the drawback of such an approach entails the overall cost of development of the measurement programme, the main benefits lie in the focus of the dedicated representation to discover a particular feature of the design which can be detected by the derivation of metrics.

The next section presents the core and final part of the interpretation process whereby the profile analysis process is explained. Naturally, it is expected that the outcome of the analysis is the suggestion of potential solutions to the design problems tackled.

### 3.4.3.3. Profile analysis and design feedback

The analysis of the results is mainly a synthesis activity. In gathering and referring back to the information found during the entire course of the measurement, the analysis of observations made from the graphical representation leads towards explanations of the phenomena observed. The profile analysis and the extraction of design feedback are closely related tasks. Simply, the former aims at discovering and explaining the profiles obtained while the latter describes the necessary design actions to be done to improve the design. Figure 3.21 represents the final phases of the measurement cycle.

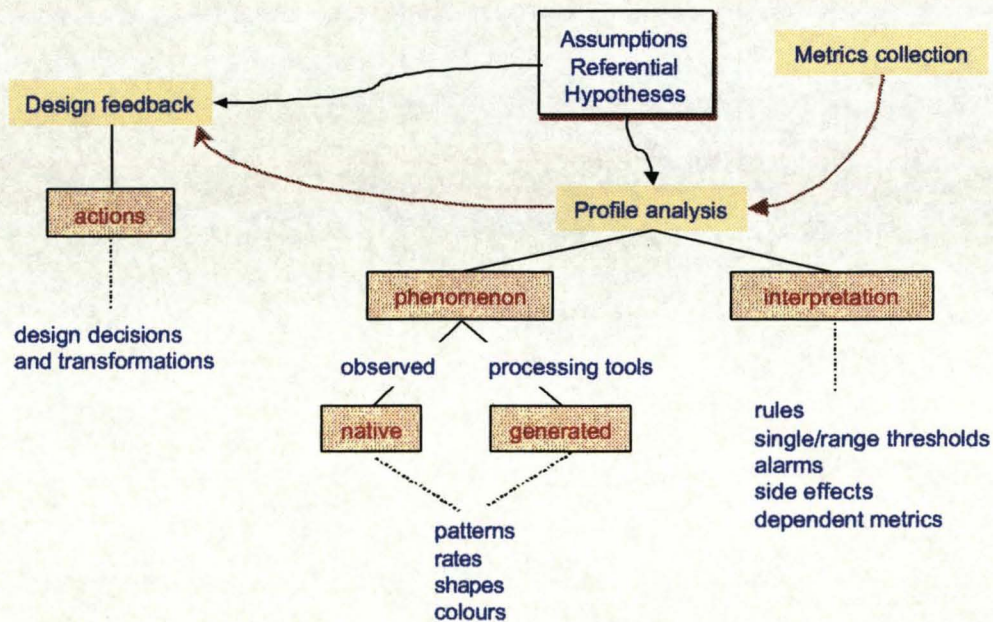


Figure 3.21: Profile analysis

John McGregor [McG95] identified three techniques to interpret metric values. The observations can be based on:

- The rate of changes of the value over iteration.
- The direction of changes of a set of values.
- The standard deviation from the mean of a set of values.

McGregor's techniques are mainly based on the observation of changes occurring to a raw data set. However, the phenomena observed on a curve can be of different nature. When a design problem is identified, it may be possible to define the conditions in which the problem occurs. In such a case, the data set may be processed before display in order to explicitly show the identified phenomenon on a curve. Therefore, the interpretation process is based on the following two factors (Figure 3.21):

- The presence of *phenomena* i.e. noticeable features, which can be either:
  - \* **Native:** without transformations, the data values exhibit particular visual characteristics e.g. peak, exponential rate of increase or decrease, minimum, maximum. Note that outstanding characteristics may be not be visually explicit e.g. not necessarily a peak on a graph. Notice that the absence of a phenomenon may be the sign of an unusual characteristic and would required further attention.
  - \* **Generated:** under some conditions, particular visual characteristics can be generated when the data is processed beforehand. For example, to obtain a macroscopic view of the data, it may be useful to show ranges of values instead of all values in a data set. This permits a reduction of the size of the data set to be displayed on a graphic, therefore facilitating its reading.
- The notion of *interpretation rules* is one possible approach to generate design feedback and suggest actions for improvement. For instance, suppose that in the context of the method redefinition profile, a threshold value of 40% is arbitrarily chosen as cut-off point. Then, if the PCRM measures show two values that are higher than the threshold on two consecutive levels of DIT, it could indicate the presence of the MDR problem. Therefore, such a situation requires the analysis of the source code to discover further information on the causes of the problem. Note that, in general, threshold values are determined by measurements done in the past in a similar context and domain. In that respect, benchmarks are commonly adopted instruments for the interpretation of metrics and determination of "goodness thresholds". Unfortunately, benchmarks are rare due to the additional cost involved in the measurement process and the relatively unpopularity and non-maturity of metrics for OO systems.

A possible definition of an interpretation rule is given as follows:

---

**Interpretation rule:**

In a given context of measurement, for a quantifiable aspect of a design attribute, an interpretation rule permits the logical deduction of the causes of the phenomena observed on a chosen representation. An interpretation rule indicates or suggests explanations on the observations of particular phenomenon for a given representation.

---

Therefore, interpretation rules constitute a mean for inferring design feedback and suggest required design actions to the designers. In Figure 3.21, the referential values are values such as threshold values, averages, minimum or maximum. They are pre-calculated or arbitrarily chosen for reference. Sometimes, subjective choices based on experience are chosen as referential values. This area is still argued amongst the research community. However, when the referential values are well identified e.g. benchmarks, they can be used within detector tools as element of comparison. In addition, to support the search for a particular phenomenon during the profile analysis, various investigation tools providing facilities for pattern searching, querying, filtering, simulation and history of profiles may be considered. Some of the tools are discussed in chapter 5.

An important characteristic of the analysis process is the influence of factors such as the assumptions, the referential values and the hypotheses defined earlier in the measurement process. The interpretation rules tackle such factors in reducing the introduction of uncontrolled factors during the interpretation. This provides a better degree of accuracy in the conclusions generated.

#### 3.4.3.4. Factors affecting the interpretation process

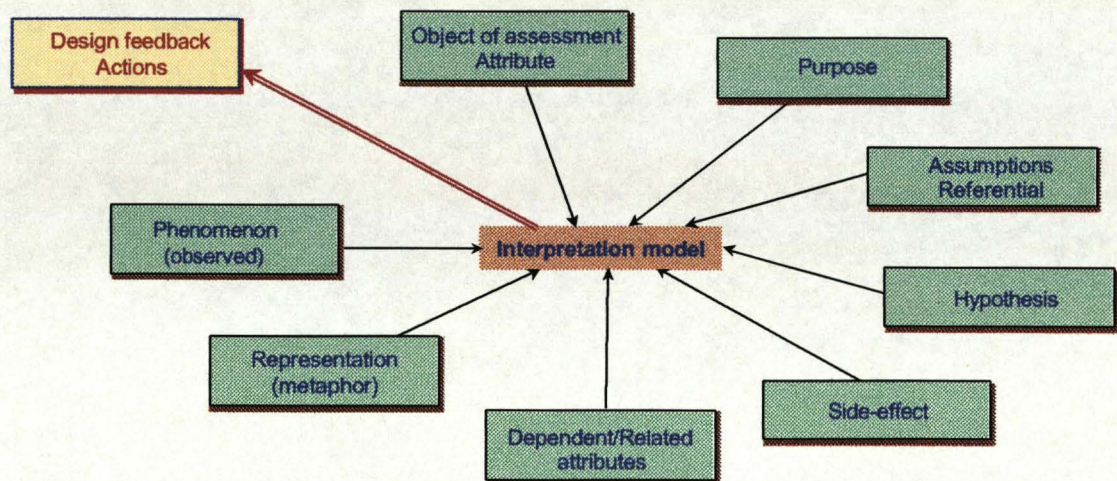


Figure 3.22: Interpretation factors

Figure 3.22 shows the factors affecting the interpretation process. Often ignored in the literature, these factors are rarely emphasised for the interpretation process. As expected, the output from an interpretation process is the generated feedback for design improvement. Interpretation rules

directly correlate causes to effects by identifying the list of factors from the different phases of the measurement (see Figure 3.22). To experimentally demonstrate the validity of the metrics, the technique of interpretation rules helps in confirming or refuting the stated hypotheses.

The side-effect factor relates to the interpretation and uses of an OO concept to solve a problem. In the case of inheritance, it has been demonstrated that various possible uses of the method redefinition mechanism affect the solution design. Often, it is the programming language features that generate unexpected designs referred to as the *side-effect* factor. The investigation of potential side effects is beneficial to the interpretation process as it provides explanations on the origins of the problems. Sometimes, the designers produce “non-conventional” designs on purpose to tackle a specific problem. For instance, the quality of the design may degrade when code optimisation is required and hacks may be utilised. If a side effect is known, then the causes of the problem may be easily understood.

The investigation of dependency relationships between attributes assessed is one possible approach for discovering the effect of one attribute on the other. When a dependency relationship exists between two attributes, the corresponding metrics are therefore dependent (see section 2.4.6.1). Interpretation techniques can then fully benefit from this observation. For example, it would be possible to discover the logical chain of events between related metric results sets to understand how the changes to an attribute affects the result of the other. Another example relates to the redefinition and the encapsulation mechanisms. If the properties of a class are declared as private, they are not visible and accessible from other classes, therefore, no redefinition is possible for the subclasses of the class. This implicit dependency relationship between metrics opens various ways of improving the assessment process. In some cases, it can be efficiently used in a predictive manner. Suppose that two metrics *m1* and *m2* are directly related, the knowledge of evolution of *m1* allows the prediction of evolution of *m2* and vice-versa.

### 3.5. Conclusion

This chapter examined the MDR problem in inheritance hierarchies and proposed a set of novel metrics for the measurement of redefinition. Details of the technical issues involved in the measurement programme were given and can be used in a more generic context. The behavioural inheritance analysis technique is a possible approach for discovering methods life histories regarding their redefinition status. Finally, a description of a metrics interpretation framework was given for tackling the problem of metrics results interpretation.

To demonstrate the benefits of the redefinition metrics, the next chapter presents a metric prototype collector tool that enables automated collection of the metrics. Details of the requirements, design and architecture are described together with some sample screen shots. Further details concerning the design of the prototype tool can be found in the Appendix.

## **4. Metric tool collector and implementation issues**

*"Not everything countable counts and not everything that counts is countable"*

– Norman E. Fenton

### **4.1. Introduction**

The availability of automated tools within a measurement programme is a necessity for the data collection phase. If metrics were to be applied manually, the task would be very exhaustive and prone to errors. As the metric collector tool examines the design information, this must be in a format recognisable by the tool. In general, the design information is available in one of the following main forms:

1. As a textual document on paper.
2. Within a CASE tool.
3. As textual files used by either a development environment or directly by a compiler.

In case 1, the use of an automated tool is not possible. Case 2 requires the knowledge of the format in which the design is stored under the CASE tool. In such cases, a possible solution is to generate the corresponding implementation in a particular language which in turn, could be processed in the same manner as in case 3. The programming code still remains a common basis from which the extraction of design information is possible. If the CASE tools do not support code generation features, a costly approach would involve the development of an integrated metrics tool within the CASE tool architecture. In case 3, additional implementation of parsing tools is required for the derivation of the metrics. For development environments providing metaclass capabilities, the design information may be directly accessible without the need for further tool development. Although features like metrics definition, metrics collection and results visualisation are desired [Bri96], the diversity of environments necessitates dedicated metric tools. In order to limit the experiments to the demonstration of the applicability and usefulness of the redefinition metric set, the following aspects guided the design and implementation of the prototype metric collector:

- A simple metric collection may be limited in functionality and use existing software applications as much as possible.
- The use of a prototyping language enables rapid development of applications.
- The identification and extraction of design information should be possible at minimum cost.
- The algorithms for the computation of the redefinition metric set should be replicable in different environment.



- Class libraries are required as subject of study.

This chapter introduces a prototype metric collector tool which:

- Automates the data collection for the redefinition metrics at class, hierarchy and system levels.
- Demonstrates possible novel representations.
- Features a method profiler for the analysis of the method's life history.
- Provides an example use of the technique of the alarmers.

The next section details the requirements for the prototype metric tool.

## 4.2. Requirements

### 4.2.1. Features

The purpose of the metric collector tool is to provide the user with a minimal set of features facilitating the derivation of the metric redefinition set described in section 3.2. The derivation process mainly consists of the automation of the data collection for further processing and analysis. The development of a metrics collector tool emphasises the fact that particular attention should be given to the feasibility and cost of such development within the measurement programme. To date, few generic metrics tools are extensible and flexible enough to permit an easy implementation of new metrics [SimLew98]. However, these are still under development.

For the purpose of this thesis, the following features of the prototype tool are considered:

- A browser which permits the display of the design to be assessed: in particular, the representation of a class hierarchy is required in order to choose sub-hierarchies for assessment.
- Implementation of the redefinition metrics algorithms.
- A method profile analysis tool that permits source code analysis for MDR problem discovery.
- Persistent storage for the metrics results. As the design process is incremental, many stable versions of the model may constitute viable solutions, therefore, it is interesting to assess these versions comparatively. Thus, a persistent facility is needed for storing previous measures. Essentially, it is desirable to be able to store metric results and other possible attributes related to the measure of the particular design subject being assessed. Therefore, persistent storage should provide a mechanism for dynamically creating objects with their associated attributes and then provide functions for retrieval of existing objects. Its underlying model is not of main importance.
- A data representation tool: the existence of powerful graphic packages on the market will suffice to satisfy the purpose of the experiments. However, the possibility of dynamic linking

between the prototype metric tool and the graphic package is envisaged as well as functionalities for creating novel representations.

- An implementation of an example of detection technique.

In addition, functionalities such as printing, exporting, importing and metrics management capabilities are also desired. Using the Smalltalk language, a rapid prototype development is possible. Furthermore, the IDE also provides support for dynamic manipulation of class hierarchies and user applications. The development is done within a PC-based environment.

### 4.3. Analysis and design of the metric collector tool

This section highlights the design issues for building the metric collector tool. The redefinition metrics set assesses the mechanism of method redefinition in a single class, in class hierarchies or in a system. For the derivation of the PRM, PRMH, PCRM and PEM metrics (see section 3.2), the internal structure of classes and existing inheritance relationship information are gathered. Before the derivation of metrics, two main questions need to be answered:

- How can one detect if a method is a newly defined method for a class?
- If a method is inherited, how can one recognise that the method is extended or redefined?

Other states for a method have been described in section 3.3.4 but not relevant to the calculation of the redefinition metrics. An immediate answer to the first question is to check if the method exists in at least one of its ancestor classes. If it exists, then the method can only be redefined, extended, reused or cancelled in the subclasses. The fact that an inherited method exists in a class i.e. presence of its signature, implies that the method is redefined, however two cases may arise: the method is replaced i.e. completely redefined, or the method is extended i.e. reuse of the inherited implementation. Note that cases of particular super calls such as directed super calls and dishinheritance (see Figure 3.15 b), section 3.3.4.1) are not considered as a valid extension of a method from a conceptual point of view.

The next section presents the problem of hierarchy parsing with regard to the derivation of the redefinition metrics at different levels.

#### 4.3.1. Class lineage and parsing strategies

Several parsing strategies can be envisaged for the search of redefined or extended methods [Mey97, Riv96, Ste90]. In single inheritance class hierarchies, the lineage of a class can be examined in localising the direct parent in a bottom-up fashion and by repeating the process for the parent class until reaching the Root class, the whole list of ancestors can be obtained. In the case of inheriting from multiple classes, the correct path is found in analysing the calls to the inherited method in the current class (see section 3.3.3).

Whether support is given by the development environment or not, the prototype metric tool is dedicated to the language studied in the sense that specific language syntax is taken into account. One of the other tasks is then to detect if a method falls under the case of extension. Consequently, it is possible to conclude that all other redefined methods are either declared as polymorphic or completely redefined.

#### Metrics derivation for the different levels

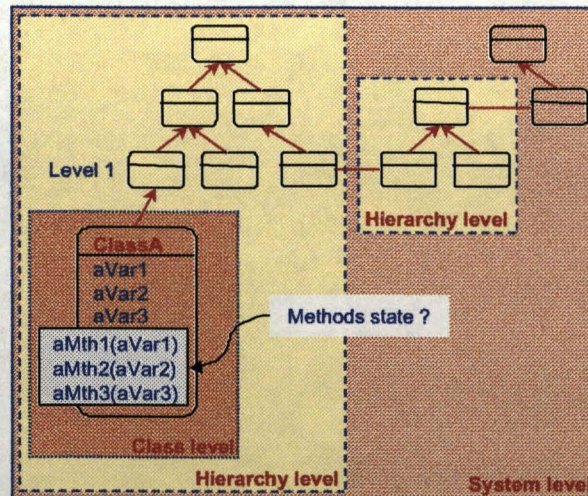


Figure 4.1: Levels of derivation

Figure 4.1 shows the classes of a system. The different shaded areas represent the three different levels for which the redefinition metrics can be applied. For each level, the list of classes to be included in the calculation is also shown. The different levels are:

- **Class level:** only the single class is concerned.
- **Hierarchy level:** the user enters the sub-root class name of the hierarchy. Then all subclasses of the sub-root are included in the computation list.
- **System level:** the user enters the list of classes in the system.

Note that at system level, the computation of the metrics does not differentiate whether classes situated at a particular level inherit from the same branch of the hierarchy or not. For example, all classes at level 1 in Figure 4.1 would be included in the calculation of the metrics although not inheriting from the same root class (further issues were described in section 5.5).

To realise the different types of searches required by the metric tool, two main parsing strategies are shown in Figure 4.2.

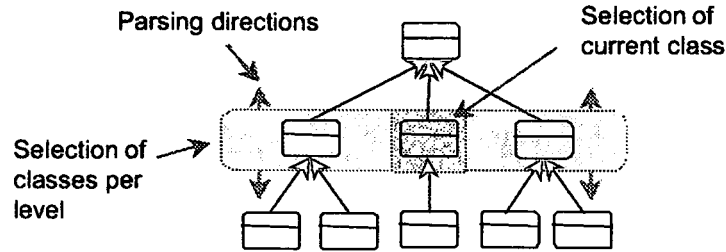


Figure 4.2: Parsing strategies in class hierarchies

The calculation of the hierarchy redefinition metrics requires the examination of classes level by level. To do so, the metric tool selects all classes situated at the same depth in the branch assessed and then, for each class at this level, examines its ancestors, its children or both depending on the information researched. The process of parsing is the main activity during the course of the derivation. The appropriate use of upward or downward parsing direction avoids unnecessary processing time. For example, in the case of the calculation of the redefinition metric for a class, downward parsing is not necessary.

For each class to be included in the calculation, an extraction of the relevant design information (section 3.3.2) from the class is done. Further details on the utilisation of the Smalltalk language for the realisation of this process can be found in the Appendix.

#### 4.4. Architecture

The design of applications with object concepts naturally separates concerns into different abstractions. Based on a Model-View-Controller (MVC) architecture, the metric collector tool encompasses three main components shown in Figure 4.3 and is entirely part of the Smalltalk environment. The Smalltalk class library<sup>19</sup> therefore provides the main development features of the language.

<sup>19</sup> The Smalltalk Express version 2.0.4 was used for the metric tool development and the creation of the user interface was done using the WindowBuilder Pro/V GUI builder [ObjSha93].

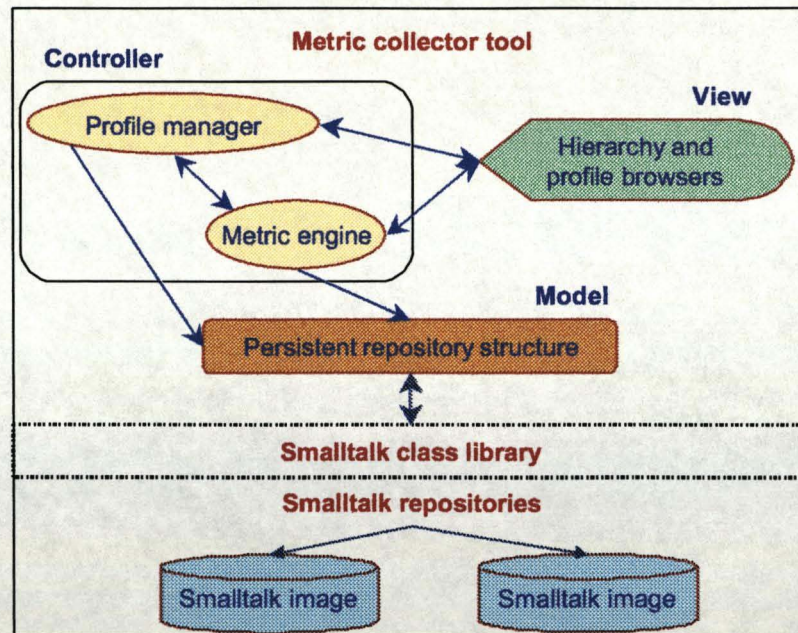


Figure 4.3: Metric collector tool architecture

The main three elements of the metric collector tool are detailed below:

- The profile manager and the metric engine act as the controllers. The profile manager ensures that the persistent repository is configured for the storage of method profile objects. The metric engine processes metric derivation requests.
- The persistent repository structure represents the data model for the method profiles. The underlying persistency mechanism relies on the Smalltalk environment and its images.
- The user interfaces include the hierarchy browser and the metric result panel.

The benefits of such an architecture for a metric collector tool lies in its simplicity and adaptability for extension of new features. Three core classes represent the three elements in the architecture. At the centre of the architecture is the profile manager object. It co-ordinates the despatching of tasks, ensures that the method profiles are created and returns the results to the display panel object. Following the object design philosophy, one important aspect of the metric tool's architecture is that the components are abstract enough to carry out their tasks independently. Smalltalk applications are "embedded" within the Smalltalk environment. The persistent repository (PR) is an adapted version of the persistence system used in [Owe95]. Represented as an additional layer on the top of the Smalltalk class library (Figure 4.3), the PR consists of a set of classes which provides capabilities for managing persistent objects within the Smalltalk image.

- The analysis and design of the components of the architecture can be found in the Appendix. However, two of the features of the metric prototype tool: *the concept of alarmers* and *the data interpretation system*, will be illustrated in chapter 5. Both features have been integrated within the hierarchy and profile browsers.

#### 4.5. User interfaces

In this section, the user interfaces illustrates the main features of the prototype tool. Decomposed in four sections: derivation, profile metric manager, method profiles and alarmers, the description of the tool covers the aspects shown in Figure 4.4.

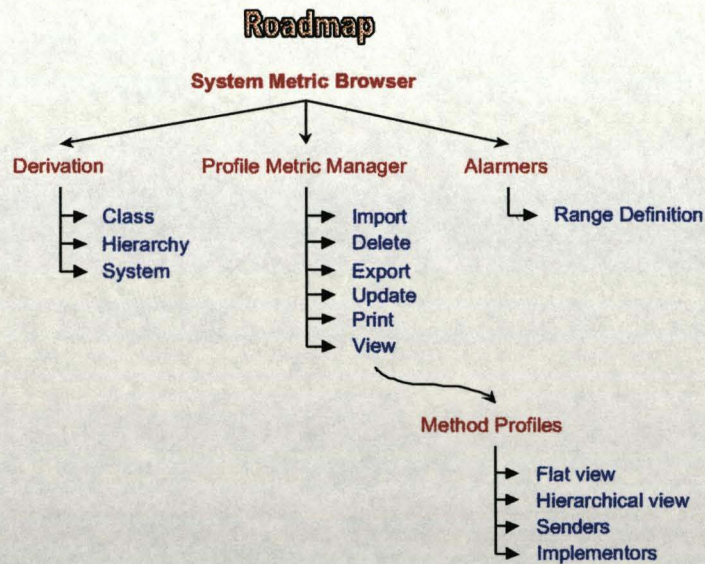


Figure 4.4: Roadmap for user interfaces presentation

In Figure 4.4, for each main feature, the available functionalities are presented as a tree. The following sections describe the metric prototype tool from a user point of view. Explanations about the derivation process and the supporting tools for analysis are given.

##### 4.5.1. The System Metric Browser

Figure 4.5 shows the main user interface for the metric prototype tool. It includes a hierarchy browser on the left-hand side panel and a tabular display of metric results on the right hand side of the window.

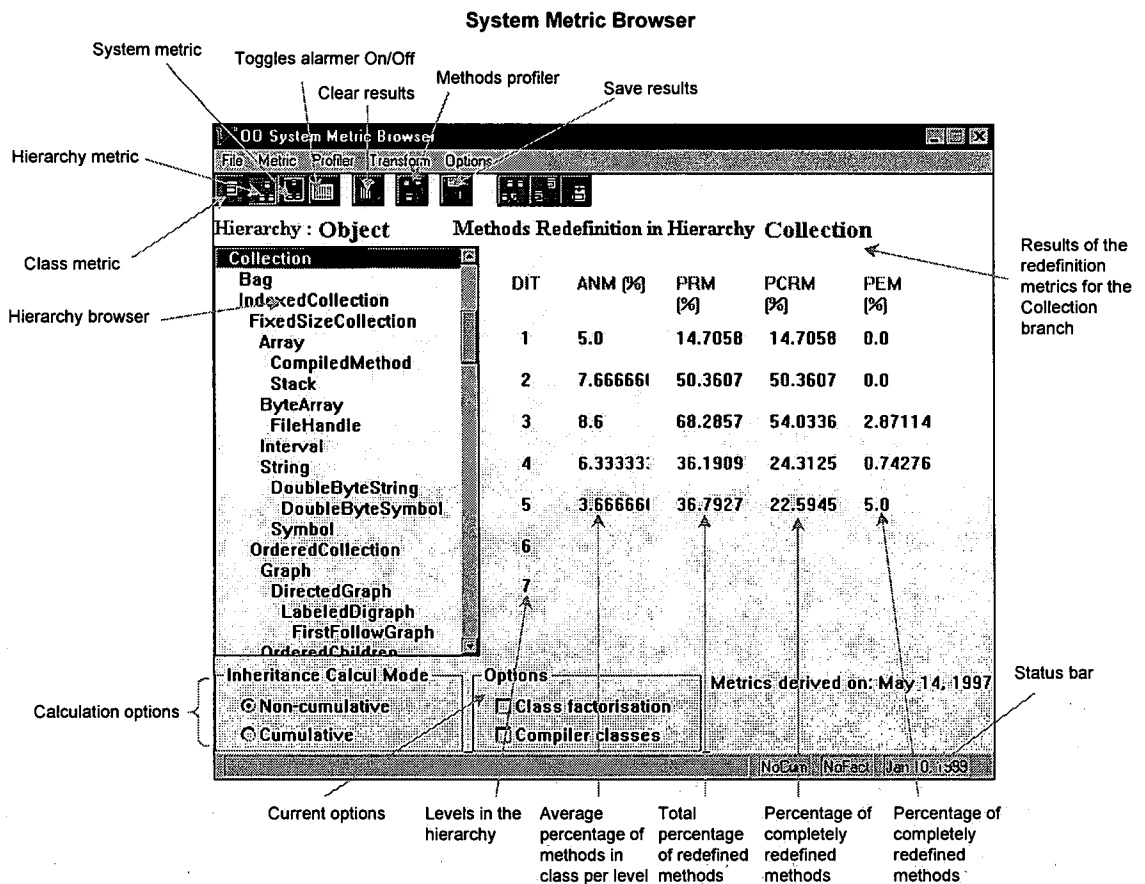


Figure 4.5: Prototype metric tool main window

The request for a class, a hierarchy or a system metric is activated either by the **Metric** menu or by the first three buttons on the tool bar. The calculation mode is set in the left-bottom panel and the chosen option is automatically reflected in the status bar. The result panel display, including the titles and metric results, are only shown after the completion of a derivation request. A function permits clearing this panel if needed. Note that this functionality only deletes the values in the browser window but not the corresponding method profile object either in memory or in the image.

For each derivation request, the date of derivation is shown above the status bar and this date is updated if a new derivation request is made on the same classes.

#### 4.5.2. Metrics derivation

In the case of a system metric request, an instance of the **SelectSystemClasses** class is created and the user is asked for the selection of classes to include in the system (Figure 4.6). The list of all classes in the Smalltalk environment is presented as a flat alphabetical list on the left-hand side of the dialog box in Figure 4.6.

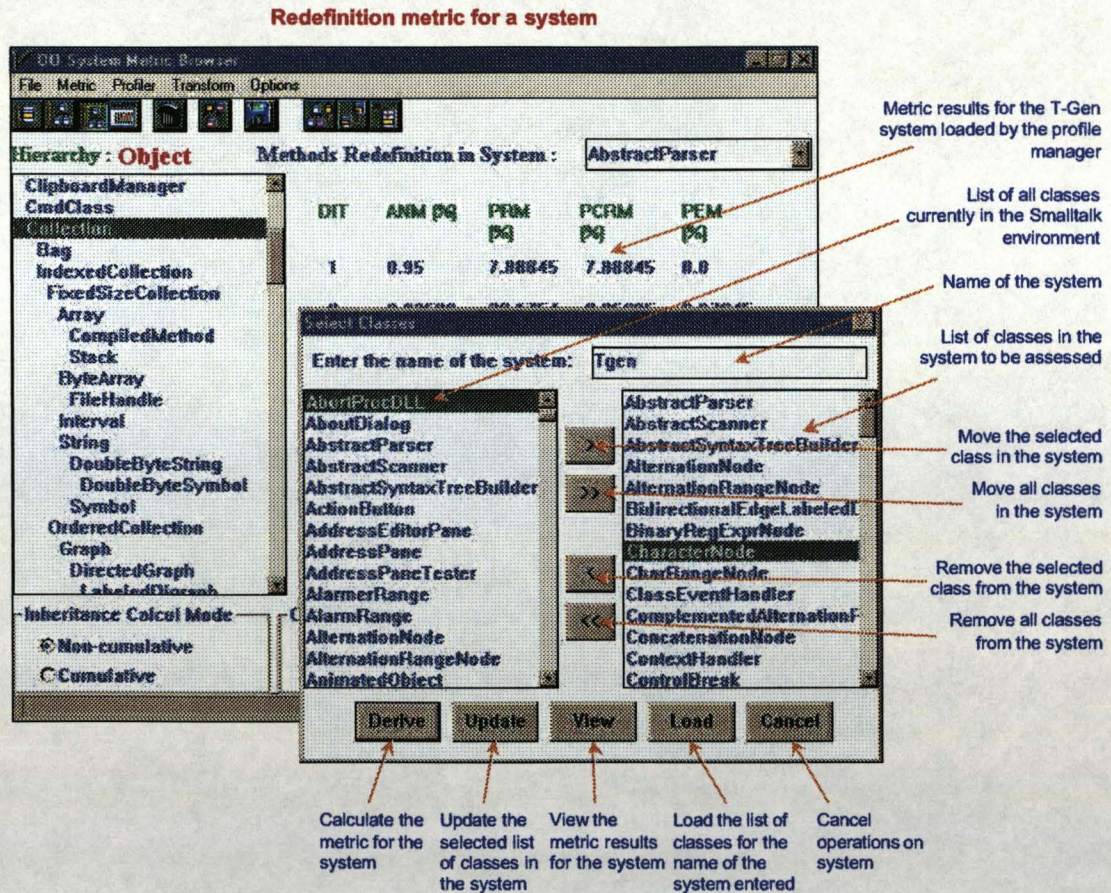


Figure 4.6: Redefinition metric at system level

A set of functions is also provided for the management of classes in the system. The metric results for a system are also stored as persistent method profile objects, thus the presence of a Load function for the reloading of classes of a system that have been previously stored. Often, it is also convenient to add or remove classes from the system as it evolves. The Update function permits a quick modification of the list of classes of the system without having to reselect all classes. The Derive function requires the computation of the metrics for the selected classes and the View function returns the stored results without re-computation of the metrics.

In the case of derivation at class or hierarchy level, the input of, respectively, a class name or a top node class name suffices for the execution of the calculation of the metrics.

#### 4.5.3. The method profiles manager

To retrieve existing metric results from the persistent repository, it is necessary to load in memory the corresponding method profile object. A list of these can be browsed using the method profile list manager in Figure 4.7. The left-hand side panel shows this list and the entry field on the right-hand side permits the manual input of the profile object name.



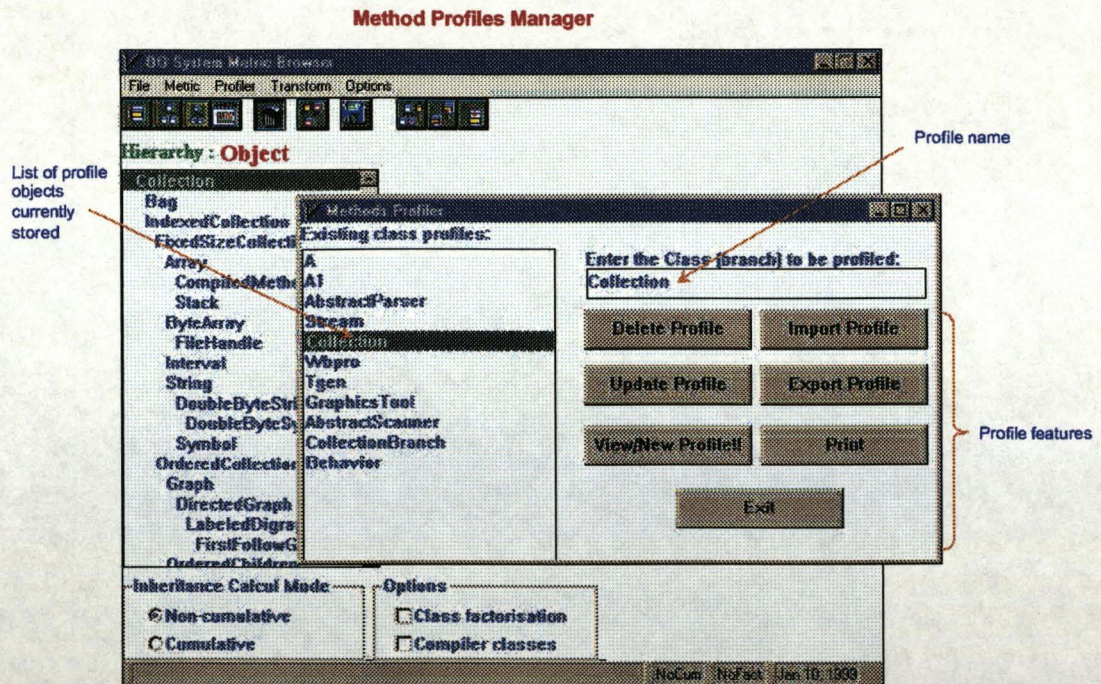


Figure 4.7: Method profile list manager

The list of features associated with the profiles is accessible via the buttons. The deletion of a profile is a physical deletion of the object from the repository. Similarly, on a request of the Update profile function, an automatic deletion of the object is done before the re-computation of the metrics. The View/New Profile option calls the method profile browser (see Figure 4.8). Note that, although the method profiles are used to store and to reload measures on a class hierarchy or a system, the activation of methods browsers is only available for measures on hierarchies. The Export profile functionality is an alternative possibility for saving a method profile. It relies on Smalltalk's object dump facility that writes a compressed description of an object along with its referenced structure on disk. An example benefit of the use of such a mechanism is that it allows saving of different versions of the same method profile objects, therefore enabling a comparative assessment of the measures. The saved files may also serve as back-ups files as well as being uploaded in another Smalltalk Express environment providing that the metric tool is available. To do so, the Import profile facility reads such binary files and permits an easy reloading of the method profiles into the repository.

Rather than directly print a method profile as the name of the Print functionality would suggest, it saves the method profiles information in textual files that can be directly reused by other applications or printed for documentation. This is particularly interesting for the processing of the results by third party applications in particular graphical applications.

## 4.5.3.1. The method profiles browser

Figure 4.8 shows the method profiles for the Collection branch. Divided in two separate panels: the upper and lower panels respectively give details about replaced and extended methods. In each panel, three windows permit the discovery of the methods' life history. The left window shows the whole list of parent classes that exist in the requested branch of the hierarchy. Then, on selection of any of the classes in this window, the set of redefined or extended methods of the selected class is displayed in the middle window. For example, in the upper panel, the list of redefined methods for the Collection class is shown. And finally, on selection of any method in the middle window, the list of subclasses of the current parent class where the method is redefined or extended in the hierarchy is shown in the right window. Thus, the method profile browser shows the details of methods' life history as described in section 3.3.5, thereby permitting the confirmation of the existence of the MDR problem in suspect classes. For example, in Figure 4.8, it can be seen that the includes: method is replaced in the branch OrderedCollection < IndexedCollection < Collection.

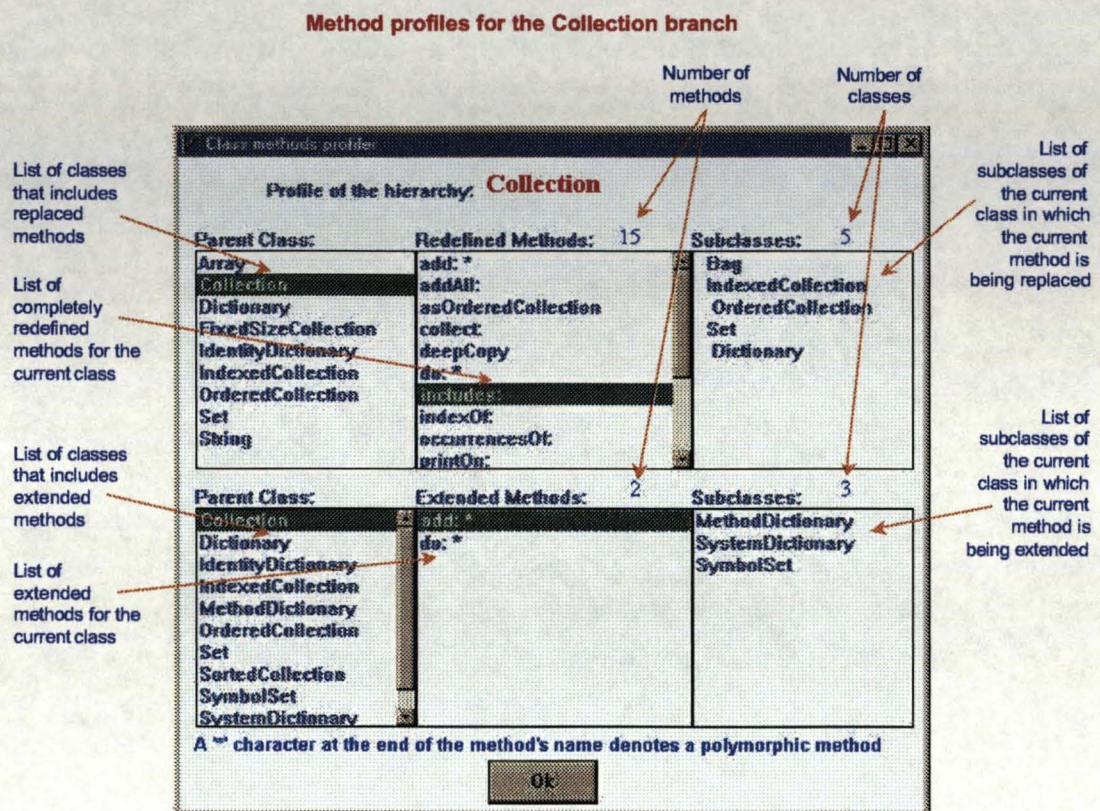


Figure 4.8: Redefined methods browser

Note that any methods that have been originally defined as polymorphic are notified by their name being followed by a '\*' character.

In both panels, it is possible to access four additional features on activation of the right mouse button on any selected method (see Figure 4.9):

- showList
- showInheritance
- Dependents
- Implementors

When a detailed search of the use of methods is needed, the first two functions may facilitate the process of interpretation. For ease of reading, the list of subclasses in the right window can be shown as a flat list or a hierarchical list e.g. in Figure 4.9, a hierarchical view of the subclasses for all add: replaced methods of the Collection class is displayed.

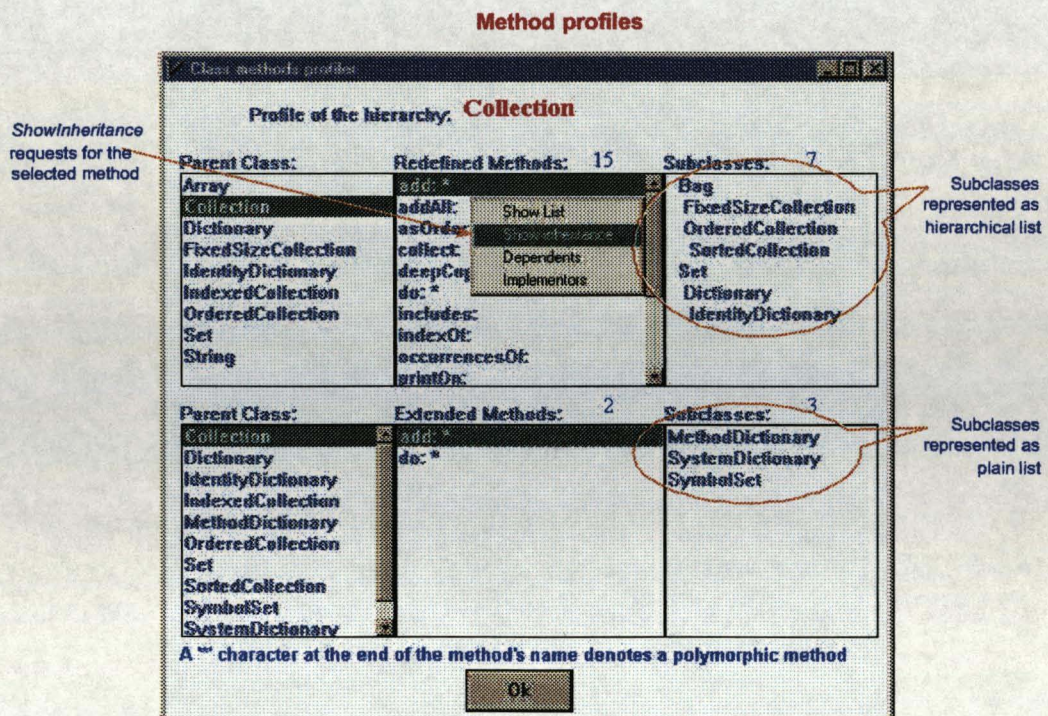


Figure 4.9: Features of the methods browser

During the course of interpretation, it is interesting to know the list of methods that refer to the method being studied. In such a case, it is possible to search for the list of classes and associated methods that refers to a method name<sup>20</sup>. For instance, in Figure 4.10, the dependent classes of the includes: method are displayed in the Method dependencies window.

<sup>20</sup> Note that in the Smalltalk terminology, methods are referred as *senders* for the reason that the method names act as the messages between two objects i.e. message-passing mechanism.

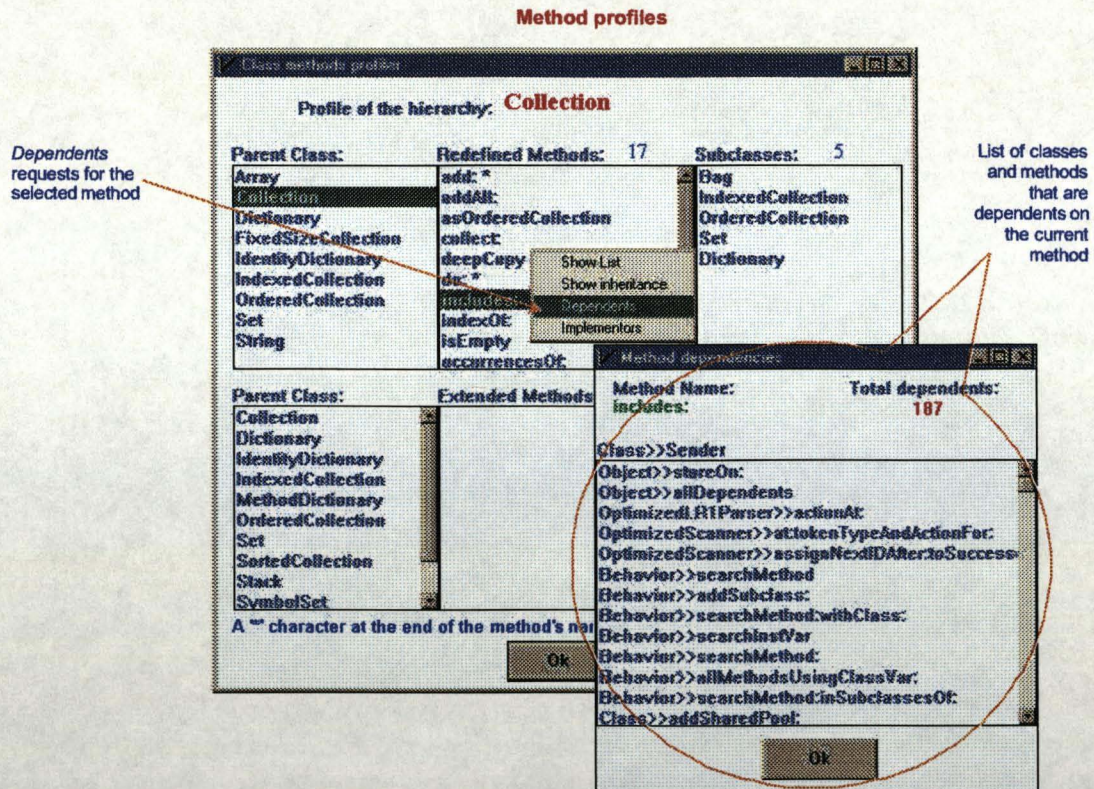


Figure 4.10: Method senders

The list of dependencies for a particular method mainly indicates how the method is being reused in other classes. In the Method dependencies window, a `class>>method` format is used to represent a method of a class that refers to the selected redefined method in the method browser. Indeed, the list of dependent classes may contain classes not in the branch of the hierarchy being assessed. The search of such dependencies for all redefined methods down the branch of the hierarchy sheds light on the various uses of the method, therefore on the reasons why it is being redefined. Also, it gives useful information if an eventual modification of the redefined methods is envisaged. Recall that in a class hierarchy, the change of an existing class or method is a difficult task, as the semantics should remain consistent with its class lineage. The complexity of change varies depending on how the class or method is referred to in other classes. For example, the total references of the `includes:` method equal 187 (Figure 4.10).

Similarly, classes that re-implement a method are referred to as Implementors. In fact, this functionality gives similar information to the right-hand side window in the method browser, however an indication of the DIT is also given by the Implementors class list in the form

DIT → Implementor

The Senders and Implementors functions can be called from both redefined and extended panels on selection of a method in the middle window.

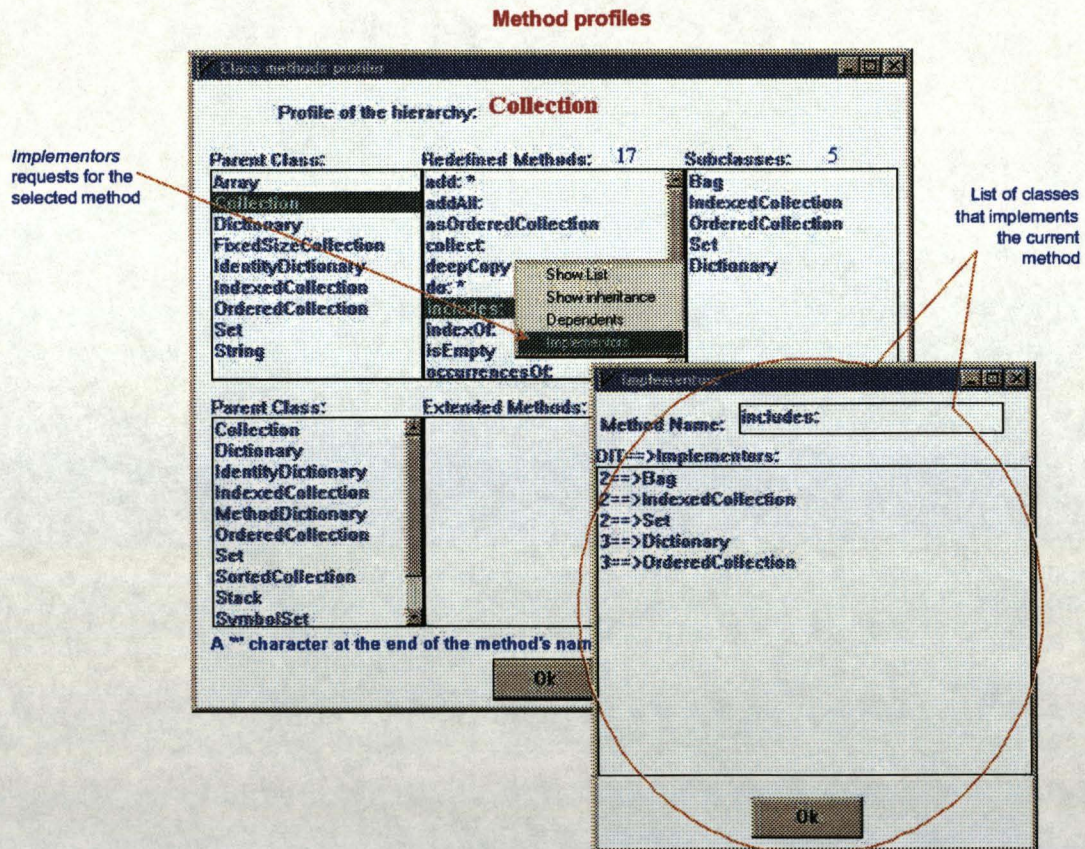


Figure 4.11: Method implementors

From an interpretation perspective, it is interesting to detect the list of method implementors within the branch of hierarchy assessed but also in other branches of the entire class library. In such a case, if the same method exists in different branches, many other issues have to be tackled such as the similarity or dissimilarity of the semantics of the method particularly if the method is being redefined in all branches. This problem constitutes other design issues that are not covered in this thesis although the discovery of such problems is possible.

#### 4.5.4. The definition of ranges for the alarmer

To set-up the ranges used by the alarmer, the `getAlarmRange:` method creates an instance of the `AlarmerRange` window that allows the user to manually input the values of the seven ranges (Figure 4.12). Recall that the entered values define seven ranges of percentages and each of the ranges corresponds to a different colour range bar.

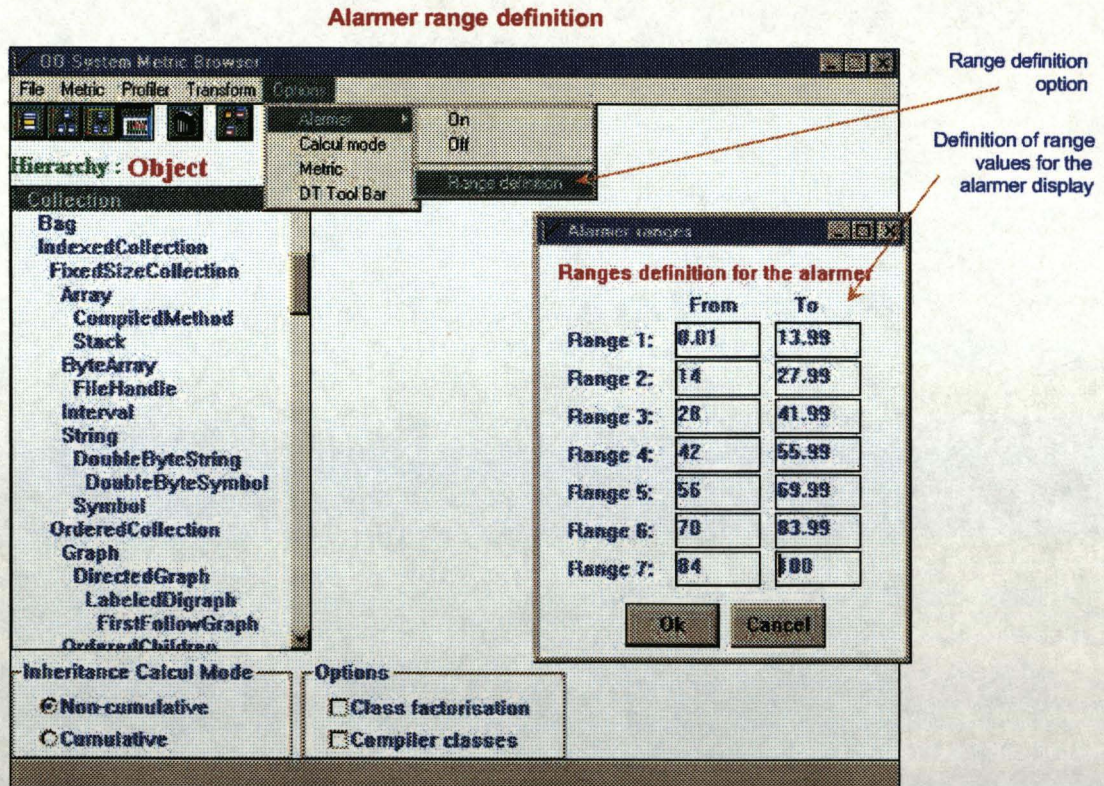


Figure 4.12: Alarmer ranges definition

Figure 4.13 shows an example of results obtained from the Collection branch. The different colour range bars are displayed directly underneath each metric value providing that the alarmer function has been tagged on (Figure 4.5). Note that the colour range bars themselves are previously defined and associated with the different ranges defined. In the current version of the tool, the range bars are bitmaps that can be redefined for different colours or shapes. However, the association is presently hard-coded for the purpose of the visualisation experiment.

### System Metric Browser with alarmer display

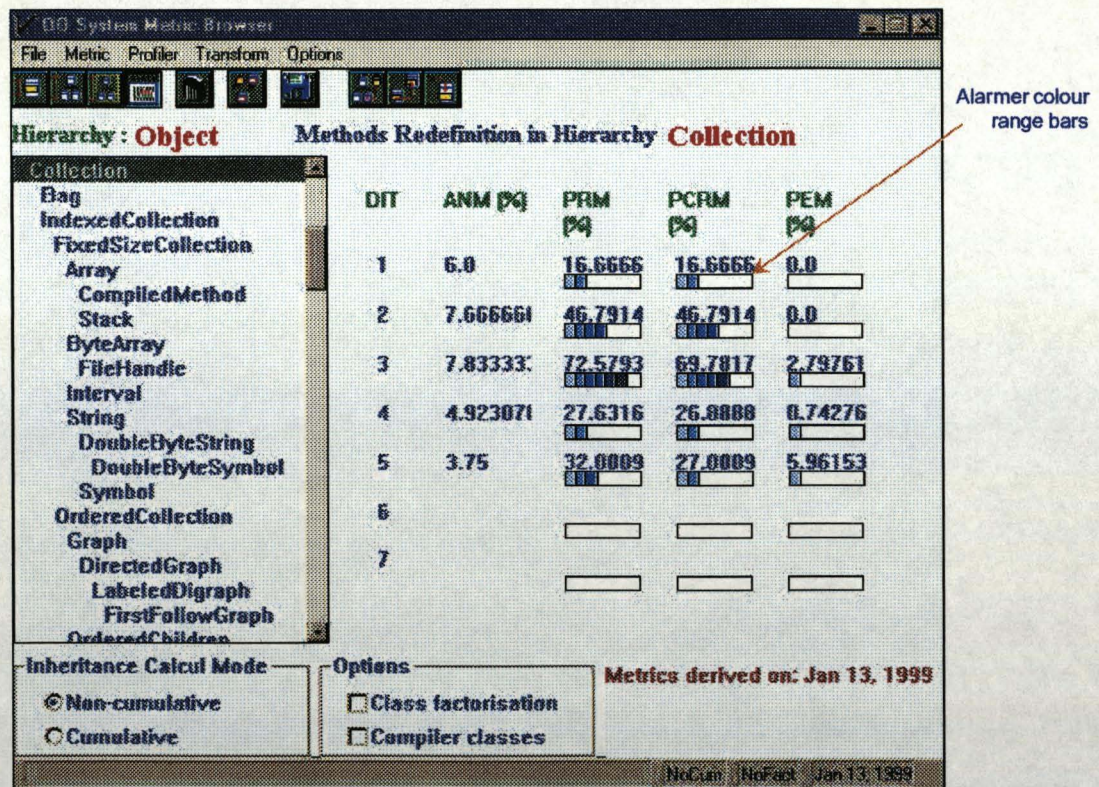


Figure 4.13: System Metric Browser with alarmer display

#### 4.6. Concluding remarks

The metric prototype tool benefits from an integrated interface where the class hierarchy component is visualised together the metrics results. Therefore, is it possible to immediately relate the analysis of the results to the relevant classes in the hierarchy. If further analysis of the hierarchy is required, the Smalltalk class hierarchy browser provides additional features. Recently, the development of such metric collector tools or code analysis tools for programming languages such as C++ or Java appears popular in industry<sup>21</sup>. As software applications are increasing rapidly in size due to the complexity of the business requirements, it seems natural that such metrics analyser tools are being developed as well.

One of the main requirements for the metric prototype tool is the importance of support provided by the development environment for both the accessibility to meta-information but also for a rapid implementation of the tool. Only the class library integrated within the Smalltalk Express

<sup>21</sup> JavaDocGen is a Java static source analysis, JavaSQA is an Object-Oriented program quality assurance tool and JavaStructure is a structure analysis and diagramming tool for Java source code. These tools are developed by International Software Automation, Inc., <http://www.softwareautomation.com>, 1999. PC-Metric for C++ (PC version) and UX-Metric for C++ (SunOS version) are source code analysis tools for C++ and are developed by SET Laboratories, Inc. P.O. Box 868 Mulino, OR 97042.

environment and the WindowBuilder Pro/V class library were required. Although portability was not an issue, the specific part of the metric tool lies in its interfaces, therefore they depend on the supporting GUI class libraries. The core classes in the remaining part of the metric tool architecture use fairly standardised functionalities that are supported in many Smalltalk flavours, therefore facilitating the portability of the tool.

An important feature of the tool without which the collection process would become rapidly cumbersome relates to the persistence of objects. This is realised with native features of the Smalltalk environment and saved within its image, all objects in the system including the method profile objects are only physically updated if an explicit **Save** command is requested or when exiting the environment. The **Save** command acts like a 'commit' command in a database in the sense that all existing objects in memory are saved in the image. Although this behaviour remains consistent with the Smalltalk procedures, it may also be constraining in some cases. For instance, if unwanted changes occur in the class library and new method profiles are expected to be saved, the changes must be undone before requesting objects to be saved. In most cases, the existing procedure is sufficient for tackling the main issues with the metrics.

Concerning the graphical representation functionalities for the metrics results, the Microsoft Excel™ 97 package was used with the exception of the implementation of the colour-coded representation within the prototype tool. By consequence, the creation of the graphical representations requires the metrics results to be transferred within the Excel worksheet. This process was manually done in the existing version of the prototype tool. Indeed, an automatic transfer would remove all the necessary manipulation. This is possible with the use of the Microsoft Object Linking and Embedding™ (OLE) technology and is envisaged as further development.

Overall, the metric prototype tool demonstrates that the redefinition metrics set is derivable. Automatic metric collection is possible at class, hierarchy and system level as expected. In addition, the implementation of a possible method profile browser gives insights on the problem of MDR, therefore generating feedback on the subject assessed. The experiments with the redefinition metrics are described in the following chapter and illustrate the applicability and benefits obtained from the analysis of the metric results.



## 5. Experiments

The aim of this experiment is to demonstrate that the redefinition metrics are derivable and produces results that may suggest potential design problems. Given the description of the metaclasses' facilities for design information extraction in section 3.3.2, the experiments were carried out on the Smalltalk Express<sup>22</sup> class library [GolRob85] and a third-party application called T-gen. The reasons behind such choices originate from the following factors: the size of the software applications or class libraries, the presence of inheritance and the availability of the source code. As the measures were taken on existing applications or class hierarchies, the design details are not known apart from a high conceptual level understanding of the subjects assessed. As a class hierarchy may cover many distinct abstractions in different branches e.g. **Collection** and **Stream** branches, it is desired to assess these different branches in isolation. By consequence, the same above-mentioned factors affected the choice of the relevant branches for assessment.

The experimentation is conducted as a five-stage process:

1. Collection of the metrics for the different branches
2. Analysis of the general PRM<sup>23</sup> metric for the different branches.
3. Analysis of the PCRMM and PEM metrics for each of the branches or system.
4. Investigation of various graphical representations for the metric results.
5. Implementation of a simple example of a detection technique called *the alarmer technique*.

This chapter demonstrates how a high level of method redefinition suggests the existence of design problems such as the MDR problem. In the first part of the experiment, only the general PRM metric is considered. The metric gives an overview of the redefinition profile for the class hierarchy. As the redefinition metrics set is a novel set of metrics, no previous results, benchmarks, thresholds or profiles exist, therefore the interpretation of the results can only be supported by the detailed analysis of the class hierarchy and the available code. Ideally, the access to design documents would shed light on the interpretation of the profiles. The shape of the curves obtained is the main guideline for interpretation. It is aimed at recognising pattern profiles that illustrates a specific aspect of the design e.g. "normal curve", "curve suggesting an MDR problem".

In the second and last part of the experiment, the previous results are further discussed with the derivation of the PCRMM and PEM metrics for the same branches. The finer-grained results i.e. ratios between the amount of replaced and extended methods, give opportunities for a better

---

<sup>22</sup>In this thesis, Smalltalk Express™ designates the version based on Smalltalk/V® Win16 and WindowBuilder® Pro/V provided by ObjectShare®, a Division of ParcPlace, <http://www.objectshare.com>

<sup>23</sup> The percentage of redefined methods (PRM) metric is obtained in calculating the PRMH for every level in the class hierarchy with linearisation of the inheritance graph i.e. no duplicates in the ancestors' list for a class.

interpretation of the design assessed. In general, when an unusual phenomenon in the profiles suggests further clarification, the designers ought to refer to the design considerations for inheritance assessment<sup>24</sup> described in section 3.2. Ultimately, references to the source code are needed in order to pinpoint precisely any potential defects.

Also, a simple detection technique called the “*alarmer*” technique is used for the identification of suspected design problems occurring under certain conditions. It is shown how the evaluation of different possible visualisations for a set of metric results not only suggested potential design problems but, depending on the type of visualisation, the same data set can reveal different characteristics.

The list of hierarchies assessed in the experiment is shown in Table 5.1. As previously stated, one of the main criteria for the choice of the hierarchies presented in the experiment relates to the number of classes in the branches or in the systems.

Type of subject assessed	No. of Classes	Description
Object hierarchy	427	Root of the Smalltalk class library and other third-party classes
WindowBuilder Pro/V system	144	GUI builder for Smalltalk Express
T-gen system	116	Lexical parser
Collection branch	25	Set of container classes
Stream branch	5	Set of Input/Output stream classes
GraphicObject branch	40	Set of classes for window management
TreNode branch	38	Subset of classes of the T-gen system
AbstractScanner branch	10	Subset of classes of the T-gen system
Object hierarchy with the T-gen system installed	549	Smalltalk and T-gen classes
Collection hierarchy with the T-gen system installed	34	Collection and T-gen classes

Table 5.1: List of assessed hierarchies

Graphical representations of the raw metrics results are generated by the Microsoft Excel97© package. On the below figures, the PRM metric is represented on the x-axis and the DIT level on the y-axis. Note that the maximum DIT shown on the graphics is 7 as no hierarchies include further levels.

### 5.1. Overview of the method redefinition profiles using the PRM metric

This first part of the experiment outlines an overview of the metric results for the selection of hierarchies described in the previous section. The initial analysis of the method profiles obtained suggests potential recognisable patterns on the use of redefinition for the assessed hierarchies. It is

---

<sup>24</sup> The method profiler in the prototype metric collector tool is an adapted version of the behavioural inheritance analysis method (section 3.3.5).

also aimed at discovering unusual characteristics in the method profiles that would suggest good or bad use of method redefinition. Overall, the grouped presentation of the results gives a 'feel' of the use of the redefinition mechanism in the hierarchy.

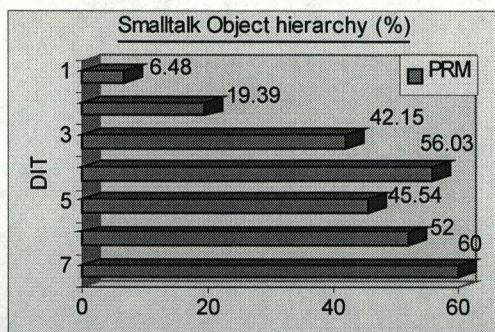


Figure 5.1: PRM for the Smalltalk Object hierarchy

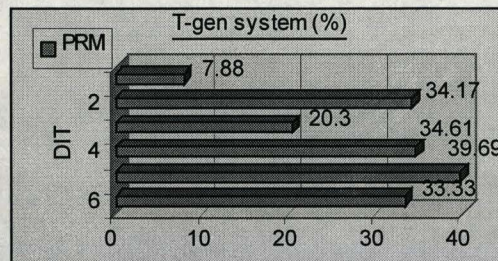
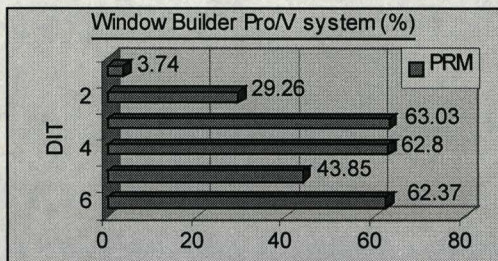


Figure 5.2 (a) and (b): PRM for the WindowBuilder Pro/V and T-gen systems

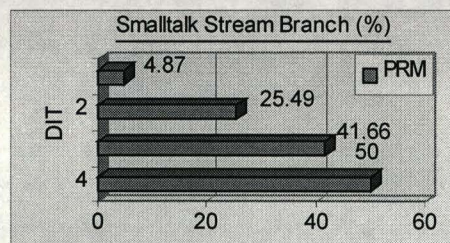
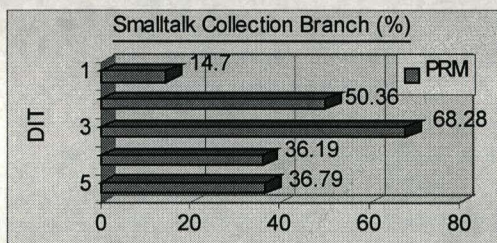


Figure 5.3 (a) and (b): PRM for the Collection and Stream branches

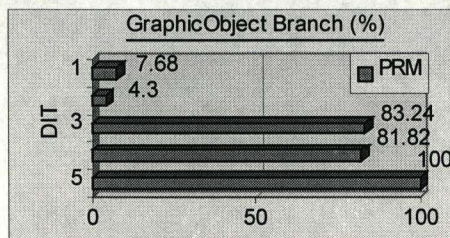


Figure 5.4: PRM for the GraphicObject branch

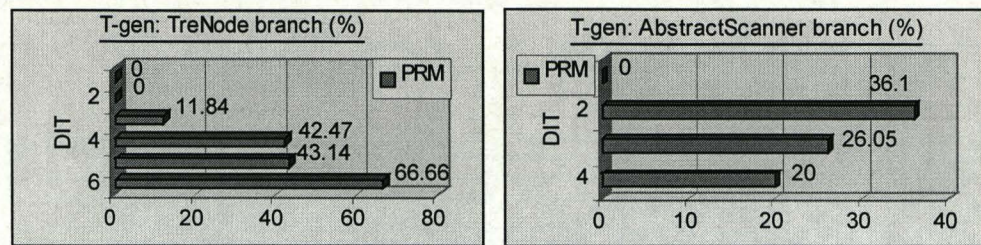


Figure 5.5 (a) and (b): PRM for the TreNode and AbstractScanner branches

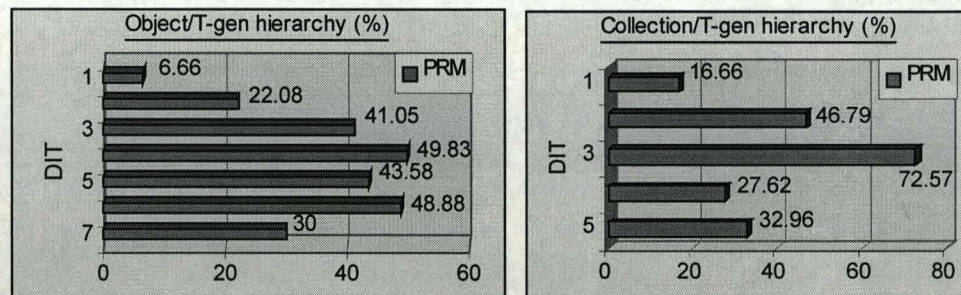


Figure 5.6 (a) and (b): PRM for the Object and Collection hierarchies with the T-gen system installed

Figure 5.1 and Figure 5.2 (a) and (b) represents the method redefinition profiles for three of the largest (> 100 classes) hierarchies assessed. Although these hierarchies are isolated for the measurement process, they constitute different systems. The other branches assessed are part of the systems.

Figure 5.3 (a) and (b) show the Collection and Stream classes redefinition metric profiles. They are generally recognised to be at the origin of similar framework of classes in other programming languages. Figure 5.4 (a), Figure 5.5 (a) and (b) show three hierarchies of smaller size (< 100 classes). The TreNode and AbstractScanner are subset of the T-gen system. In Figure 5.6 (a) and (b), the method profiles for the Object and Collection hierarchy show the metric results calculated with the presence of the T-gen system in the Smalltalk environment.

A common pattern that appears in the profiles is that the amount of method redefinition rapidly increases in the first three levels of the hierarchy, then remains stable for two or three levels and finally decreases or increases in the bottom levels. As the highest values occur in the middle or bottom levels of the hierarchy, it indicates that the core redefinition activity is located at these levels. In the first levels of smaller size hierarchies, it is noticeable that the redefinition activity is low or even non-existent. Generally speaking, it seems normal that the redefinition activity would increase as the subclasses are specialised i.e. use of abstraction. This can be explained by the fact that deeper levels of the hierarchy should include a higher number of classes and as the number of inherited methods are accumulated at each level, they are also likely to be either used or redefined.

Naturally, the first overview of the redefinition activity calls for further investigation of the low and peak values. The following sections give a deeper analysis of the metric results. For each of

the above hierarchies or systems, it is shown how the examination of the high values guides the analysis of the results to the discovery of unclear design situations. The presence of MDR is highlighted in most cases. To do so, the PCRМ and PEM metrics is derived on the same hierarchies and illustrations of a pragmatic approach to the problem of localisation of defect classes in the design are given.

## 5.2. Smalltalk Object hierarchy

The Object branch represents the whole class hierarchy (single-rooted hierarchy) which comprises 425 classes. The two curves for the PCRМ and PEM metric<sup>25</sup> enable a clear separation between two types of method redefinitions: extension and replacement. Surprisingly, most of the methods are replaced instead of being extended.

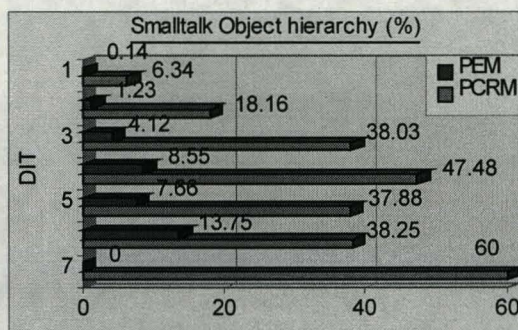


Figure 5.7: PCRМ and PEM for the Object hierarchy

In Figure 5.1, the PRМ rate of increase of the Object branch is fairly smooth. The first surprising feature (Figure 5.7) is the relatively high number of completely redefined methods (PCRМ) in the whole Smalltalk hierarchy. In this initial measure of the Smalltalk redefinition profile, from DIT=1 to DIT=3, starting with a value of 6.34% for the PCRМ, the value more than doubles in the subsequent levels denoting a strong redefinition activity. From DIT=3, PCRМ=38.03%, the next values seem to stabilise until DIT=6 although there is an unusual peak at DIT= 4 with 47.48%. Clearly, the midlevels of the Smalltalk hierarchy yield most of the redefined methods. It is argued that deeper hierarchies may generate a redefinition activity as high as the one presented in the experiment. In general, large branches such as the Object branch tend to lessen the discovery of potential problems. This is due to a leverage phenomenon when a large number of classes are involved in a measure.

<sup>25</sup> Note that the profile for the Smalltalk Object hierarchy in section 5.6 (a) slightly varies from the profile shown in section 3.1.1, Figure 3.2. The differences of measures obtained are mainly due to the evolution of the prototype metric collector between the two sets of experiments. Indeed, the prototype also lives in the Smalltalk environment, thus influencing the results. The correctness of the metrics results remains consistent as long as the same version of the prototype is included when assessing various aspects of the hierarchy.

Although recommended (see section 2.1.1), more levels implies more abstracted classes spread over more complex branches of the hierarchy making it difficult to control inheritance. This is also true for the use of the extension mechanism. If a hierarchy already encompasses many levels of inheritance, finding what the abstract classes and methods are, before the addition of new features, is a necessary and cumbersome task. The need for design aid tools to alleviate some of the designer's task is then a requirement in the modelling process. In Figure 5.7, note that the low level of PEM (13.75%) at DIT=6 is also its maximum. The interesting characteristic of the PEM values is that it has a fairly constant increase which indicates a good sign of the use of inheritance. However, at DIT=7, 60% of the methods are replaced while 0% is extended. This contradicts the essence of inheritance. Redefinition, which is recommended to be used with care, occurs frequently at all levels in the hierarchy, and extension, which is recommended, is rarely used. This raises the question of the correctness of the behavioural inheritance design.

In order to further understand the phenomena observed on the curve, it is necessary to consult the classes present in the hierarchy and the state of their associated methods (see section 3.3.5). Note that the Smalltalk class hierarchy comprises of many branches dealing with different aspects of a generic class library, therefore the results obtained in Figure 5.7 includes classes that may not be related to each other although part of the hierarchy. The overview of the method redefinition for the Smalltalk hierarchy sheds light on the way redefinition is done down the hierarchy. However, to identify the possible reasons for such profile, it is more appropriate to derive the metrics on a smaller portion of the class hierarchy. In such a way, the measures are done on classes that participate in the same abstraction. Therefore, the results are not disturbed by the effect of other classes that not related to the subject assessed.

The following experiments present the isolated branches of the Smalltalk hierarchies.

### 5.3. Collection branch and Stream branch

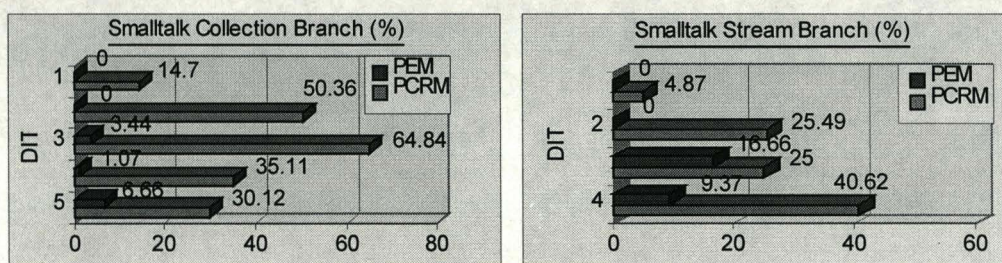


Figure 5.8 (a) and (b): PCRMs and PEMs for the Collection and Stream hierarchies

The Collection classes in Smalltalk have been well studied by many researchers [Coo92, GolRob85, Lew95a] and are particularly known for the conceptual design problems occurring in leaf classes (see section 3.1.1). A major problem concerns the amount of cancellation of property inheritance in leaf classes. Smalltalk's inheritance scoping control permits a class to stop the visibility and accessibility of a method to its subclasses in redefining the method with a body

containing the code `self shouldNotImplement`. This situation is often recognised as a source of bad design. The derivation of the redefinition metric would include the case of cancellation of properties of a class. Indeed, the precise localisation of the faulty class requires code inspection.

In Figure 5.8 (a) and (b), at  $DIT=2$ , no methods are extended. A simple explanation is that all classes at level 2 have realised the abstract methods, which is normal. The metric profiles illustrate a case where a peak in a curve permits the discovery of classes highly suspect as they present an unusual level of redefinition. For example, supposing that a threshold of 40% of method redefinition should raise an alarm to potential design defects, it would be necessary to take a closer look at the peaks happening at  $DIT=3$  in Figure 5.8 (a) and  $DIT=4$  in Figure 5.8 (b). A simple way would be to derive the PCRMM metric for each class of the concerned level. In Figure 5.9, it can be seen that the `FixedSizeCollection` class holds 100% of methods completely redefined. Such a result is unusual as none of the parent classes is declared as abstract. Although the percentage of deferred methods is not shown in the figure, the above-mentioned class seems to be wrongly subclassed. With the help of the method profiler tool, it is possible to study the hierarchy further. For instance, Figure 5.10 shows the method profile for the `Collection` branch. The `add:` method of the `Collection` class is being replaced in many subclasses (right hand side panel) situated at different levels of the hierarchy, thus illustrating a case of MDR problem. In the bottom panels, it is also shown that the `add:` method is only extended in three of the `Collection` subclasses.

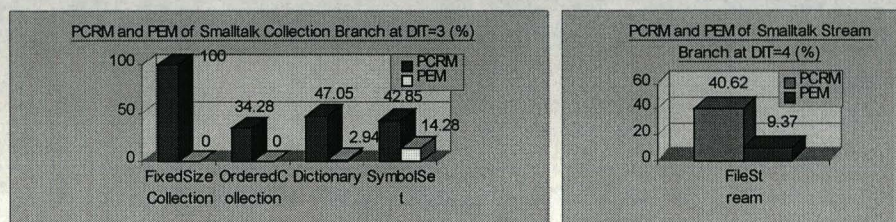


Figure 5.9 (a) and (b): Collection branch at  $DIT = 3$  and FileStream at  $DIT=4$

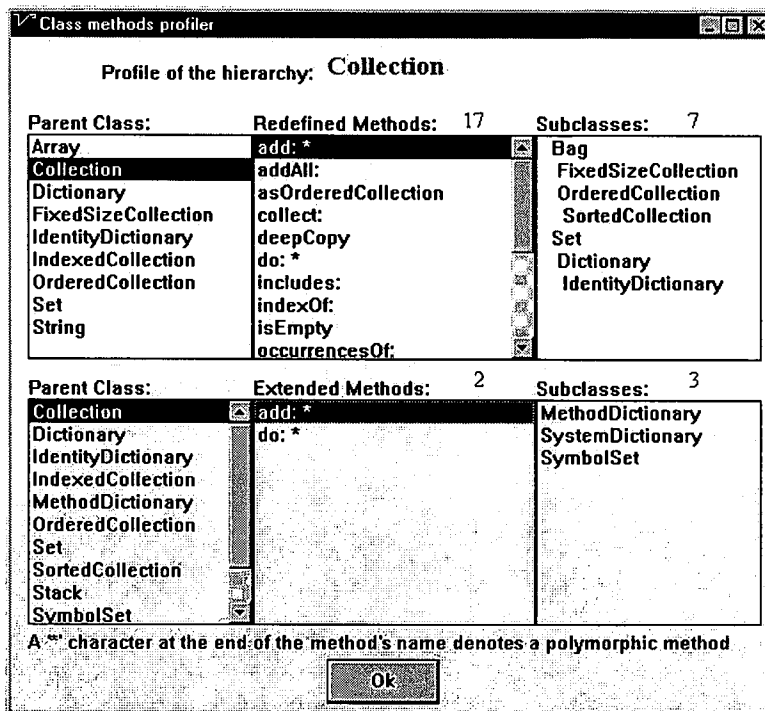


Figure 5.10: Collection method profile

The PCRM for the Stream branch (Figure 5.8(b) and Figure 5.9 (b)) is high with 40.62% at DIT=4, which represents a factor increase of 60% from the previous level. This confirms the Smalltalk Stream branch's generally recognised design defect. Due to the single inheritance scheme, the ReadWriteStream class inherits only from the WriteStream class. There is a duplication and redefinition of methods from the ReadStream to WriteStream. Note that the use of the method profiler for this branch is not shown but it also reveals several cases of MDR.

### 5.4. WindowBuilder Pro/V branch

WindowBuilder Pro/V is a GUI builder for Smalltalk/V [ObjSha93]. The tool permits the creation of the user interface including all of the powerful and standard UI elements. In addition to being entirely visual, the tool generates the necessary Smalltalk code once the design is done. A full installation of WindowBuilder Pro/V includes 144 classes. As the prototype metric collector tool was built with it, the measures taken for the Object branch included the WindowBuilder Pro/V classes as well as the prototype collector classes.

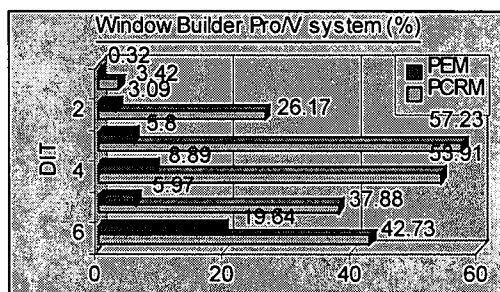


Figure 5.11: PCRM and PEM for the WindowBuilder Pro/V



GUI builders are now well established with many proprietary products such as *The BISS AWT Framework* [Bis97], *XForms* [ZhaOve97], *PowerBuilder®* [Pow98]. All of them are based on basic interface elements such as windows, scroll bars, text boxes, list boxes, radio buttons. Due to the advent of graphical development environments, it is generally recognised that GUI builders cover the essential needs of a large range of information systems. Therefore, the design of the GUI builder itself ought to be abstracted enough to achieve such requirements, thereby showing a fairly high redefinition activity as in Figure 5.11. It is noticeable that the highest measures of redefinition occur at mid-levels of the hierarchy (DIT=3 and DIT=4) rather than in top levels as previously seen for the Collection branch (Figure 5.8 (a)). Although the PCRm decreases on deeper levels of the hierarchy (DIT=5 and DIT=6), it remains fairly high with 37.88% and 42.73% respectively. On the contrary, the PEM ratio is steadily increasing down the hierarchy which suggests that inheritance is correctly used for specialising the hierarchy by addition of new features. However, recall that the measures shown on Figure 5.11 are general to the WindowBuilder Pro/V system. Complete redefinition or extension may be found only on some branches of the system and not others. A behavioural inheritance analysis for each isolated path would permit the discovery of further details of the design.

The next section describes the measures taken for the GraphicObject branch which is part of the WindowBuilder Pro/V application.

#### 5.4.1. GraphicObject branch

The GraphicObject branch is one of the largest branches of the WindowBuilder Pro/V application. It includes 40 control interface classes which permit the definition of radio buttons, check boxes, list boxes, entry fields.

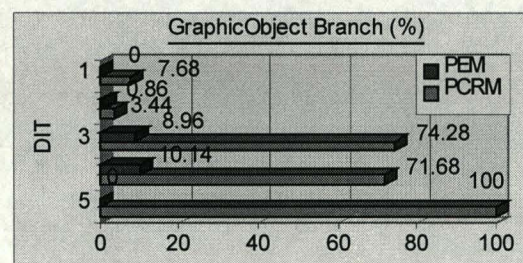


Figure 5.12: PCRm and PEM for GraphicObject branch

Figure 5.12 shows that the first two levels of the branch contains a low amount of PCRm and PEM. GraphicObject is the only class situated at DIT=1, so is the InterfaceObject class at DIT=2. Such a profile indicates that the two DIT classes provide all the necessary behaviour for future subclasses, thus the low level of redefinition. The PCRm increases by a factor of 21.6 from DIT=2 to DIT=3. This shows that method redefinition occurred at the top of the hierarchy, and questions whether the methods were initially well abstracted. For DIT=3 and DIT=4, the PCRm

are respectively equal to 74.28% and 71.68%. Considering that this branch provides all the necessary basic user interfaces elements for windows management, it is expected that most of the methods in the top classes would be redefined. In addition, each of the interface elements would be very specialised, therefore including a large amount of methods for reuse by a new application. A detailed analysis of the classes at DIT=3 is given in section 5.6. A suspect feature is depicted at DIT=5 in Figure 5.12 with PCRM=100 and PEM=0. Considering this level in the hierarchy, it is surprising that no methods were reused nor extended and that no addition of new methods were made. The study of the GraphicObject branch method profile (Figure 5.13) reveals that this phenomenon seems to happen relatively often and concerns a few leaf classes i.e. a single class in this case. Also, it is possible to detect that many methods present a case of MDR such as the drawFrameWith:at: method which is defined in the FrameObject<sup>26</sup> class

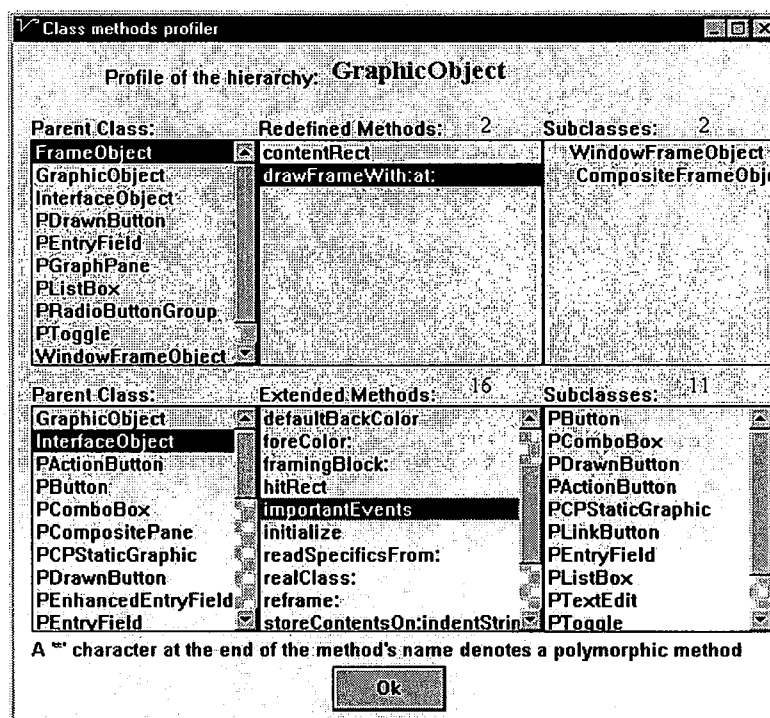


Figure 5.13: GraphicObject method profile

Note that the bottom panels of Figure 5.13 shows the list of methods of the InterfaceObject class that are being extended in its subclasses producing a PEM=8.96% at DIT=3 and PEM=10.14% at DIT=4.

### 5.5. T-gen system

*"T-gen is a general-purpose object-oriented tool for the automatic generation of string-to-object translators. It is written in Smalltalk and lives in the Smalltalk programming environment. T-gen supports the generation of both top-down (LL) and bottom-up (LR) parsers, which will*

<sup>26</sup> The FrameObject class is situated at DIT=3.

*automatically generate derivation trees, abstract syntax trees, or arbitrary Smalltalk objects. The simple specification syntax and graphical user interface enhance the learning, comprehension, and usefulness of T-gen.*" -- Justin O. Graver [Gra92]. T-gen is made of 116 classes with a maximum depth of six for the TreNode branch. As the system is a lexical and syntactical parser, most of the processing does not involve user interaction apart from defining a grammar as input. As for any other Smalltalk applications, the installation of T-gen classes, in a general sense, extends the Smalltalk class hierarchy. Similarly, the redefinition metrics prototype tool (See chapter 0) application classes are also part of the Smalltalk image. With the Smalltalk environment, many applications can live in the same image and not interfere with each other. However, assessing the redefinition mechanism of a system raises some issues concerning the choice of classes to be included in the derivation of the metrics:

- **Isolated classes:** the assessment of inheritance is relevant when, by definition, an inheritance relationship is defined between two targeted classes. If an assessment of application classes that inherit from the Smalltalk environment is desired, the question is to know whether the latter classes should be included in the derivation of the metrics. Recall that a branch of a hierarchy can be identified by locating the top node of the branch, thereby the assessment of such a branch will examine all possible inheritance paths from the top node class. As the redefinition metrics assess inheritance level by level, a first approach will only consider the application classes in the calculation. In such a way, the results obtained from the derivation of the metrics would only concern the targeted application. A second approach for deriving the metrics is to consider the whole Smalltalk hierarchy with the application classes installed, so a comparison would be possible with the original Smalltalk environment.

Isolated classes in an application raise the problem of their inclusion on the calculation of the redefinition profile for the whole system. For instance, in T-gen, the class `Graph` inherits from the `OrderedCollection` class, the class `Stack` inherits from the `Array` class, the class `ItemSet` inherits from the class `Set`, etc. `OrderedCollection`, `Array` and `Set` are part of the Smalltalk library. In most cases, isolated classes are leaf classes, therefore a measure of redefinition for a class is one possible solution. In Figure 5.14, the `ItemSet` class has `PCRM=100%` and `PEM=0%`. Although this result may suggest a design problem at first sight, the detailed study of its methods reveals that the only three methods in the class: `=`, `hash` and `isItemSet` are originally defined in the `Object` class and are not previously redefined in its intermediate parent classes `Collection` and `Set` classes. Thus, the `ItemSet` class should not be considered as suspect.

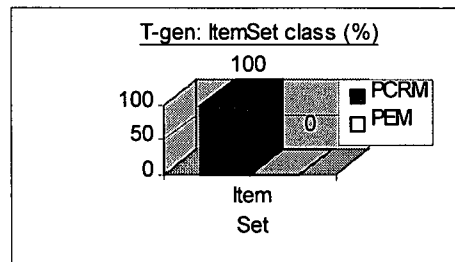


Figure 5.14: T-gen: ItemSet class redefinition profile

- Foreign classes** i.e. classes which belong to the existing library. In general, application classes extend many existing branches of a hierarchy. Suppose that the application classes derive from an existing class which has itself many superclasses in the same branch. Many ancestor classes may act as top node of a branch for hierarchy assessment. For the derivation of the metrics, the issue is to decide whether to include the parent classes or not. In such cases, there are two possibilities; including the direct parent only or previous parent classes. In both cases a mixture of classes from the existing library and the application classes are included in the calculation. This remains consistent in the sense that an assessment of inheritance is desired, thus the inclusion of all classes which act as a superclass in a particular branch. Note that inherited methods in a class are not necessarily originally defined in the direct parent class but in ancestor classes of more abstracted levels as well. The case of the Object class is special as it represents the root class (see section 2.1.3). Indeed, when the metrics are applied on the whole Smalltalk class library, the Object class is the top node of the branch. The disadvantage of including foreign classes in the calculation is that it may affect the values of the results when the proportion of foreign classes is much higher than the application classes. In a cumulative approach, this may invalidate the results in making negligible the effect of the application classes and their properties (see section 5.6) on the metric results.

#### 5.5.1. T-gen system redefinition profile

The classes in the T-gen system are spread over many different branches of the Smalltalk hierarchy. The T-gen system is made of distinct small size hierarchies with the Object class as a parent class and isolated classes inheriting from the Smalltalk class library. The derivation of the redefinition metrics is done in the same way as for the derivation on a single branch of the hierarchy. In fact, in the calculation of the metrics, classes are processed according to their superclasses, subclasses and the DIT level they belong to. Isolated classes of a system are included in the calculation of the metrics as any other classes in the system. In Figure 5.15 a redefinition profile is represented. In this experiment, the calculation is done on the application classes only i.e. no inclusion of foreign classes.

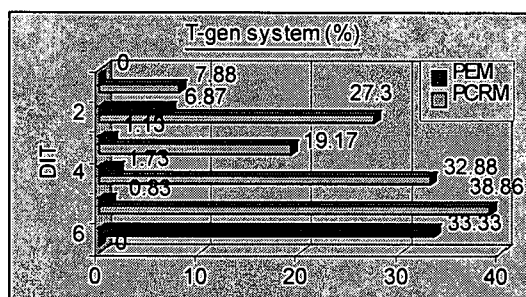


Figure 5.15: PCR and PEM for the T-gen system

Figure 5.15 reveals that, at DIT=6, 33.33% of the methods are extended but none are replaced. Concerning the PEM curve, the values remain quite low except the presence of the peak at DIT=6. Contrary to previous experiments, at DIT=2 the PEM reaches 6.87% after being nil at DIT=1. As many Smalltalk branches are involved in the T-gen system, no satisfactory conclusions can be drawn at this point. Again, the measures of the redefinition on a whole system raise the problem of interpretation. Further investigation for more detailed measures and knowledge about methods profile are necessary before suggesting any recommendations for improvement. However, it is still possible to notice that the level of completely redefined methods is high which suggests possible presence of the MDR problem in the system.

As for the Smalltalk class hierarchy, in the next sections, relevant branches of the T-gen system have been profiled and presented for further understanding on the use of the redefinition mechanism. Indeed, selected branches ought to have many levels of inheritance in order to be able to analyse the behavioural aspect of the branches.

### 5.5.2. T-gen: TreNode branch redefinition profile

The TreNode branch is the deepest branch in the T-gen system with a maximum DIT=6.

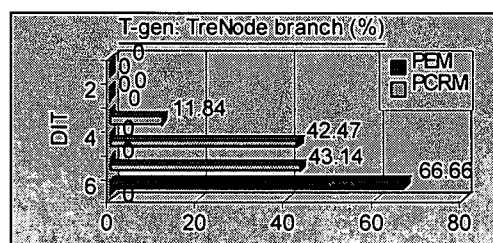


Figure 5.16: T-gen: PCR and PEM for the TreNode branch

In Figure 5.16, no metrics values were found for DIT=1 and DIT=2. Simply, it means that no redefinition has been found for classes situated at the two first levels. At the first level, an explanation of such a situation is that the TreNode class is the top node of the branch and has the Object class as its superclass. Therefore, as the TreNode class itself should provide generic methods for its subclasses, it acts as a supplier class. In addition, there was no need to redefine

inherited methods from the `Object` class, thus the nil values. At DIT=2, only a single class exists, the `ParseTreeNode` class with four methods defined as non-applicable to instances of the class i.e. the body of the methods contains:

```
self shouldNotImplement
```

The above body declaration does not impose any conditions on the subclasses of the class but only on instances of the class. No invocations of the declared methods are allowed by instances of the class. If such a situation happens, the system redirects a `doesNotUnderstand: walkback` error to the sender of the message meaning that an object received a message that it cannot resolve. As no implementations are provided for the methods, the `ParseTreeNode` class acts as an abstract class, however, in such a case, the methods should have been declared abstract as well, with a body containing:

```
self subclassResponsibility
```

or

```
self implementedBySubclass
```

As expected, subclasses of the `ParseTreeNode` class do provide the implementation for the four methods. Despite the fact that the original author's intention of prohibiting the creation of instances of an abstract class is correct, abstract methods are seen as a preferred design technique to ensure the coherence of inheritance.

Although at DIT=3, the PCRМ is low 11.84% (Figure 5.16), an investigation of the classes situated at this level reveals that three classes exist: `GrammarParseTreeNode`, `TokenSpecParseNode` and `RegularExpressionNode`. Looking at the comment for the first two classes, the author considered them as abstract classes, however, no methods were declared in those classes. This situation is typically the case where inheritance is used as a mechanism for separation of concerns more than for the intended mechanism. This does not invalidate the use of inheritance in this case; on the contrary, its use was probably intended for future development of the hierarchy. At DIT=4 and DIT=5, the PCRМ is quite high with 42.47% and 43.14% respectively. Again, when reaching the bottom classes two phenomena can be expected in a hierarchy: either the high level of PCRМ or PEM. Again, the method profile for the `TreNode` branch (Figure 5.17) permitted the localisation of suspect classes containing MDR problems e.g. the `ParseTreeNode` class at DIT=2.

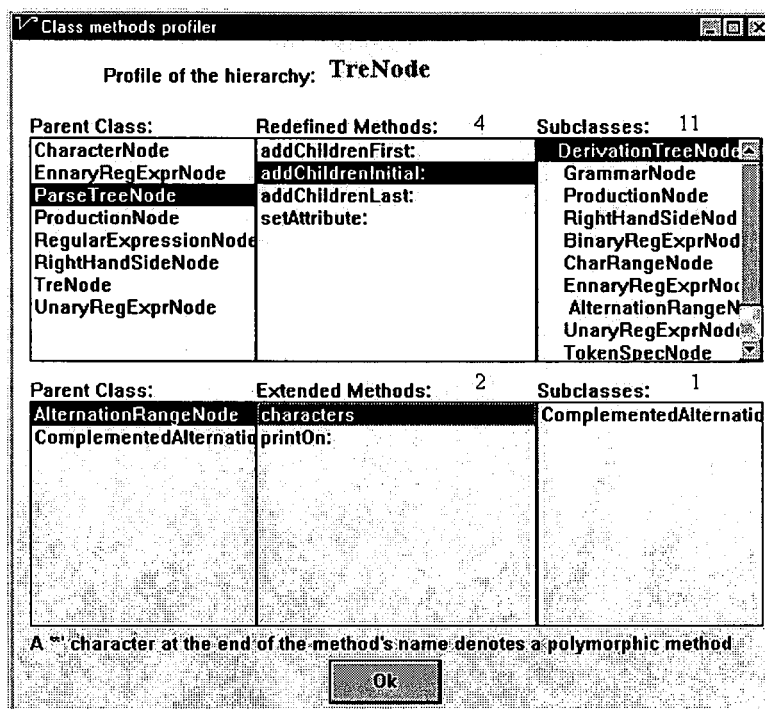


Figure 5.17: TreNode method profile

None of the methods were extended in the first five levels of the branch, then PEM is equal to 66.66% at DIT=6. As is often the case, only few leaf classes exist at deeper levels in the hierarchy. In turn, this raises the level of PCRM or PEM. Here, only a single class realises the amount of PEM (bottom panels of Figure 5.17). This example illustrates the difficulty of designing classes that extend the system behavioural capabilities rather than using the redefinition technique for realising the necessary functionalities. It has been generally recognised that, designing a well-abstracted hierarchy with use of redefinition for extension requires extra effort from the designers. Such a task is difficult to realise for the reasons that forward planning of future enhancement is necessary; however, this is, unfortunately, unknown in most cases. By nature, requirements are likely to evolve with respect of the business needs. This may be not predictable.

### 5.5.3. T-gen: AbstractScanner branch redefinition profile

The AbstractScanner branch is another example where no redefinition occurs at DIT=1. This branch is composed of ten classes on four levels of depth. A peculiarity in Figure 5.18 is that the redefinition level is constantly decreasing down the hierarchy. In order to better analyse and interpret such results, a detailed analysis of the behavioural inheritance is required. A high level of redefinition should always raise suspicions about the design but does not necessarily imply an incorrect use of the mechanism for all the sub-branches of a branch. Recall that the decision that a design is bad or good depends on the elements of comparison. For example, consider the three measures for the branches AbstractScanner (Figure 5.18), TreNode (Figure 5.17) and for the overall system (Figure 5.15). The AbstractScanner and TreNode branches are the largest

branches in the system. At DIT=2, as no redefinition activity is taking place in Figure 5.17 and PCRM=27.3% in Figure 5.15, it seems that the AbstractScanner branch is responsible for nearly all the redefinition activity with PCRM=27.08%.

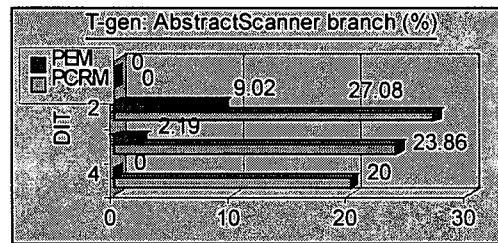


Figure 5.18: T-gen: PCRM and PEM for the AbstractScanner branch

Further investigations done with the method profile for the AbstractScanner branch confirms the presence of the MDR problem (Figure 5.19) e.g. scanToken method in the AbstractScanner class.

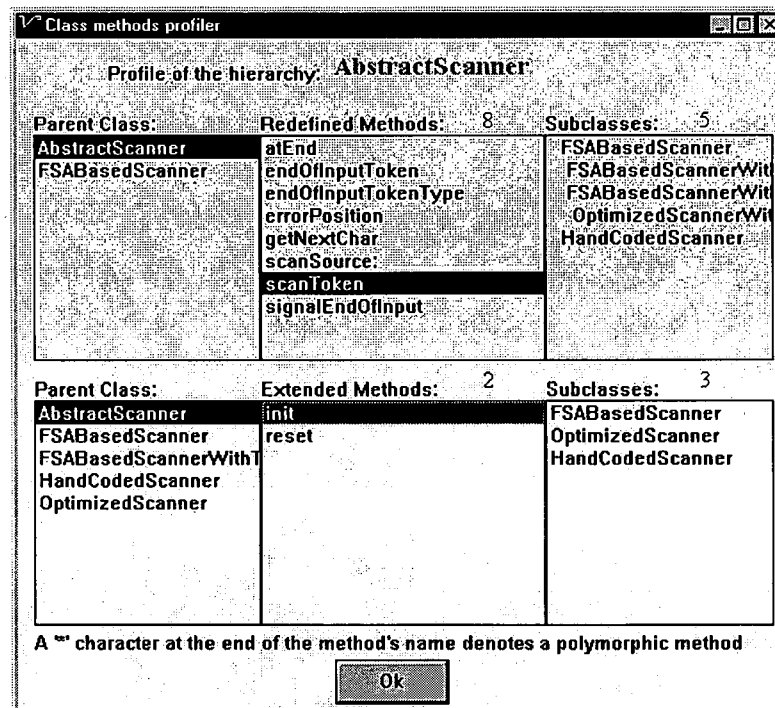


Figure 5.19: AbstractScanner method profile

Another reason to carry out such investigations is that attention should be given to the derivation of the metrics level by level and the leverage effect of classes situated at the same level. For example, an analysis of the result PCRM=20% at DIT=4 (Figure 5.18) may suggest an acceptable level of redefinition. However, when examining closely the design at this level, two classes `OptimizedScannerWithOneTokenLookAhead` and `OptimizedScannerWithTwoTokenLookAhead` exist. In the latter, the class is empty i.e. no properties are defined, which suggests that the author planned its development for the future. Thus,



it is possible to conclude that the former class has, in fact, a PCRM=40%, which makes the class more suspect.

In this example of use of metrics, it is shown that the analysis and interpretation of the metrics results still require the support of additional design or contextual information e.g. source code, to reach a viable explanation and potential solution to a design problem.

The next experiment investigates the use of the cumulative PRM for three branches of the Smalltalk hierarchy.

### 5.6. Cumulative measure for the Collection, Stream, Object and GraphicObject branches

The second approach for the calculation of the PRMC' metric i.e. cumulative metric (section 3.2.2) relates to the number of potential methods available to a class. If all inherited methods as well as the new ones defined in a class were to be considered, the accumulation of methods is likely to increase for classes situated near the bottom of the hierarchy. An experiment done on the Collection hierarchy is shown in Figure 5.20.

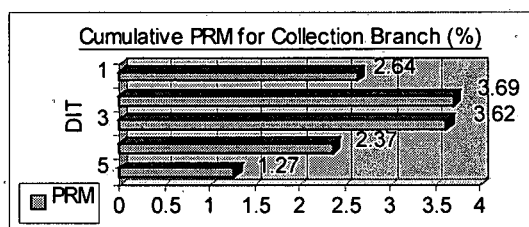


Figure 5.20: Cumulative PRM for the Collection branch

As expected, the values for the PRM metric remain low and even decrease. The Collection class is situated at DIT=1 and inherits the 155 methods of its parent Object class, giving a PRM=2.64%. From DIT=3, the PRM decreases. This is due to the fact that most of the classes in the hierarchy are situated within the first three levels. Figure 5.21 represents the number of classes per DIT level. Recall that the single root Object class is at DIT=0. The total number of classes in the hierarchy is 427. Clearly, more than half of the total classes are located nearer the top of the hierarchy. Therefore, this suggests that, per DIT level, the number of methods may be higher near the top than the bottom.

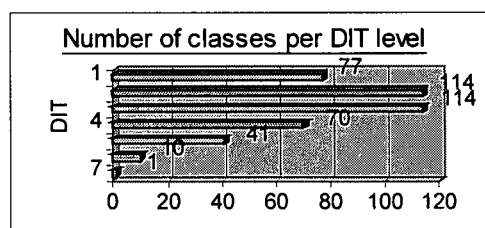


Figure 5.21: Number of classes per DIT level

From DIT=3 to DIT=7, the rate of decrease of number of classes is quite high (nearly or over 50%) from one level to the next. Indeed, the above measures only give an idea of the profile for the whole hierarchy; however, it shows that the hierarchy tends to have a “shallow shape” rather than a recommended “deep shape” [Fir95].

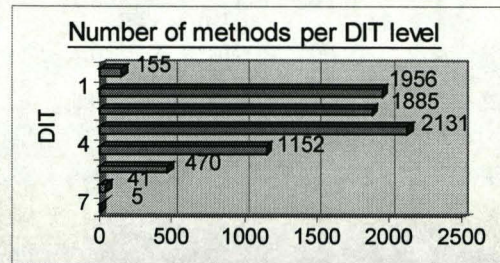


Figure 5.22: Number of methods per DIT level

Figure 5.22 shows an overview of methods per DIT level. As previously expected, the majority of methods are situated in classes near the top of the hierarchy. The root `Object` class (DIT=0) contains 155 methods. It is noticeable that for DIT=1, the number of methods is 1956 while at DIT=2, it is only 1885 although the former level contains 67% less classes than the latter level. This confirms that, in general, top classes usually contain more methods than bottom classes. It also reflects the fact that more abstracted methods may exist in the first level of the hierarchy. Thus, for each inheritance path, a portion of this high number of methods in top classes is inherited in subclasses giving a low level of redefinition when considering the accumulation of potentially available methods in the calculation (Figure 5.20).

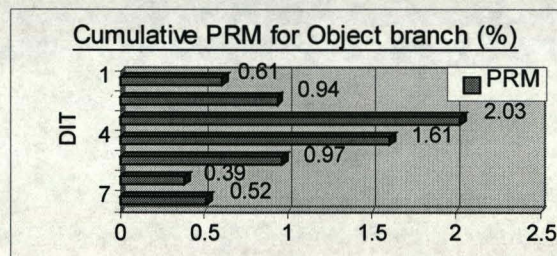


Figure 5.23: Cumulative PRM for the Object branch

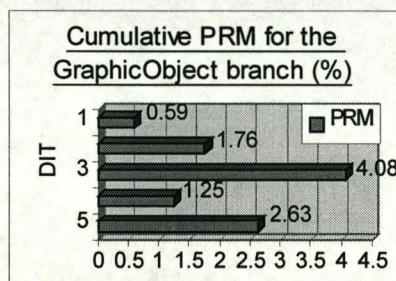


Figure 5.24: Cumulative PRM for the GraphicObject branch

Figure 5.23 and Figure 5.24 respectively represent the cumulative PRM for the whole Smalltalk `Object` hierarchy and the `GraphicObject` branches. The `GraphicObject` branch contains classes

related to GUI definition. Similarly, the values of the metric remain low. However, a similarity in the profiles seems reproduced in the different measures. All the cumulative measures have a maximum value occurring near the DIT=3 level which suggests that classes located at such level are critical classes as the redefinition activity increases to its maximum value. For the half bottom part of the hierarchy, the redefinition activity decreases due to the amount of inherited methods in bottom classes.

As a general guideline, a high redefinition activity at one level in comparison to other levels indicates that many leaf classes may exist at the concerned level, requiring the redefinition of inherited methods. Therefore, there are potential design problems. A refined measure of redefinition would then indicate the ratio between replaced, cancelled or extended methods.

The cumulative measure of redefinition is useful when considered, at a level  $l$ , with:

- The number of methods per classes.
- The number of classes.

If applied on an isolated branch of the hierarchy, a peak in the redefinition profile suggests either:

- A high number of abstract methods in top classes.
- Wrong use of inheritance at the level where the peak occurs.

For instance, in Figure 5.24 for the `GraphicObject` branch, it is clear that at DIT=3, the high level of redefinition activity is remarkable and asks for further investigation. As the measure was done following a cumulative approach, consideration should be given to the number of potentially available methods per class (Figure 5.22) when interpreting the results.

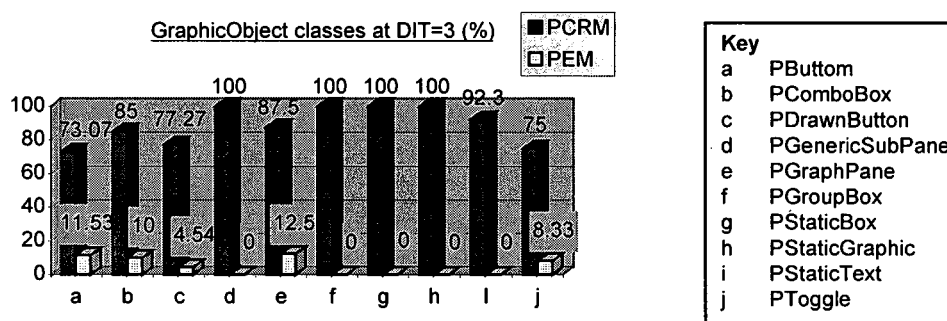


Figure 5.25: Subset of `GraphicObject` subclasses branch at DIT=3

In order to understand why the redefinition activity rises at DIT=3 for the `GraphicObject` branch, an investigation of classes situated at this level is done (Figure 5.25). The redefinition metrics is then applied on a selected subset of classes (10 out of 21) which are relevant to the demonstration. The list of class names is given in the above legend. All the represented classes contain a level of PCRM above 73% and a PEM below 12.5%. Four of the classes redefine all their methods giving a PCRM=100% and PEM=0%. A detailed method life history would then pinpoint problems such as the MDR. In this particular branch, none of the methods have been initially defined as

polymorphic. This should raise the suspicion alarm for the designers about the correctness of the classes and properties.

The following three experiments describe other interesting measures that shed light on the use of the method redefinition mechanism. In particular, focus is given to the discovery of suspect classes and the influence of method redefinition in systems that are “embedded” in a class hierarchy.

### 5.7. Effects of the T-gen system on the Smalltalk hierarchy

A Smalltalk application is tightly coupled to the Smalltalk class hierarchy in the sense that the applications classes derive from the existing class library, thereby becoming part of the hierarchy. It is then interesting to investigate the effects produced by the presence of a system in the Smalltalk environment from an inheritance assessment perspective. After installation of the T-gen application, the new redefinition profile for the Smalltalk Object hierarchy is as follows (to be compared with results in Figure 5.7):

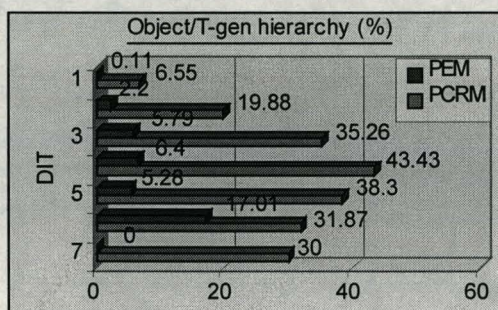


Figure 5.26: Smalltalk Object hierarchy with the T-gen system installed

The T-gen system does not seem to have much effect on the Smalltalk redefinition profile. A slight increase of the values is noticeable for the first three levels. Then for the deeper levels, the values of the PCRM decrease due to the leverage effect of less completely redefined methods in the T-gen classes for the levels concerned. Similarly, the values of the PEM still increase and are slightly higher for the first six levels and remain at zero at the seventh. Note that, at DIT=6 PEM=17.01% which represent an increase of 23% compared to its initial value. This seems directly related to the amount of PEM in Figure 5.16 for the TreNode branch.

Overall, for development environments similar to the Smalltalk environment, knowledge of the redefinition profile is interesting as it is affected by the following reasons:

- **Flexibility for development:** direct modification of the code of the native class library is possible. For instance, extension of existing classes and methods from the Smalltalk hierarchy is common practice. Indeed, this assumes that the code is available for modification. The specialisation of code to suit a developed application is a natural and valid process from a software engineering point of view. The drawback for the Smalltalk hierarchy is that it becomes more specialised, which may generate problems when more than one independent application

requires to live in the same Smalltalk image. In such cases, careful precautions must be taken in order to avoid the overriding of methods used by both applications. Usually, the delivery of Smalltalk applications is done per image, thus avoiding the problem. In languages such as C++, the native class hierarchies are provided as is. Only extension by new class addition is possible and only the interface functions are described without code availability.

- **A stand alone image:** reuse and specialisation. Whether the ratio of newly introduced classes of an application to the native classes of the library is none, low or high, the effects of the application classes on the redefinition profile completely depends on the design. Predictions of the profile depending on the shape of the hierarchy are difficult. However, if the application class ratio is high e.g. over 50%, the chances of increased dependency level is higher, thus affecting the overall class hierarchy redefinition profile. In the case of the T-gen system, the ratio is:

$$\text{application class ratio} = \frac{\text{number of classes of the system}}{\text{number of classes of the native class hierarchy}}$$

$$\text{T - gen class ratio} = \frac{116}{432} = 26.85\%$$

In comparison, the reuse ratio  $U$  [Hen96] and specialisation ratio  $S$  (see chapter 2, section 2.4.6.1 for the interpretation of these metrics) are equal to:

$$U = \frac{\text{number of superclasses}}{\text{total number of classes}} \quad U \text{ for T - gen} = \frac{153}{549} = 27.86\%$$

$$S = \frac{\text{number of subclasses}}{\text{number of superclasses}} \quad S \text{ for T - gen} = \frac{548}{153} = 3.58$$

While 26.85% of the classes are T-gen classes, the reuse ratio is 27.86% which indicates a shallow depth and a large number of leaf classes. The specialisation ratio is 3.58. According to Henderson-Sellers [Hen96], ratio values of  $U$  and  $S$  near 1 suggest a poor design which is not the case of the above values. Although, T-gen has slightly increased the level of PCRM, it has also contributed towards a “better” extension profile and a leverage effect on the whole hierarchy.

### 5.8. Effects of the T-gen system on the Collection branch redefinition profile

In general, the Collection branch is one of the branches mostly used by applications as it provides all the facilities for container management. It is then interesting to repeat the previous experiment on this branch to detect any eventual effects of the T-gen classes on the redefinition profile. The initial measures of the PCRM and PEM are shown in Figure 5.8.

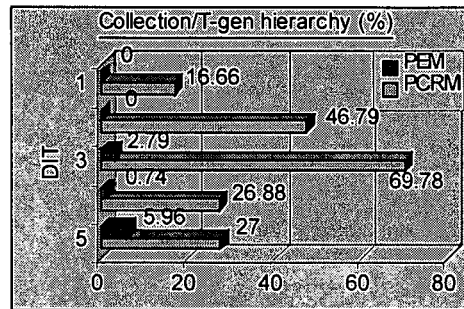


Figure 5.27 (a) and (b): PCRM and PEM for the Collection hierarchy with the T-gen system installed

Figure 5.27 shows the new profile for the Collection hierarchy. Compared to the profile without the T-gen system installed (Figure 5.8), no remarkable differences can be observed. With the T-gen system installed, the values for the redefinition metrics seem to slightly decrease apart from DIT=3. Nonetheless, each value of the extension profile decreased as opposed to what had been previously seen for the whole Smalltalk hierarchy. From the profile, the effects of the T-gen appear negligible.

### Conclusions on the first three stages of the experiments

In any assessment technique, it is important to consider the characteristic's context i.e. any factors directly or indirectly related to the characteristic, in addition to the characteristic itself and its eventual influence on other characteristics. Often, to analyse results from a metric, it is necessary to refer to other metric results to infer any conclusions, design anomalies or directions for solutions to a problem (see chapter 2, section 2.4.6.1). As mentioned in section 5.5, some design choices may involve a modification of the class library from which the application derives. Depending on the modifications, the assessment of the redefinition mechanism and inheritance in general raises other issues concerning the derivation algorithm. Design modifications concerning the behavioural aspect of inheritance may be categorised as follows:

- Insertion of a new class as an intermediate parent class. In rare cases, an identification of a new abstraction may require the addition of a new class in the middle of an already existing branch rather than adding the new class as a leaf class.
- Modification of code in the existing methods of the class library. This is not generally recommended unless there is detailed knowledge of the implications of the changes for the hierarchy.
- Addition or update of new classes or methods to the classes library. This is one of the most common tasks occurring during design. Depending on how abstract the method is, its addition may take place at any level of the hierarchy.
- Deletion of classes and methods from the class library is not recommended although possible.

The first two points involve a high level of risk of compromising the conformance of classes to their ancestor classes. Addition, deletion or update of classes or methods may have consequences on all subsequent subclasses in the branch. In all cases, the designer must verify that the implications of the modifications do not jeopardise the coherence of the inheritance hierarchy.

The issues concerning the assessment of new classes added to the class hierarchy has already been discussed in the introduction of section 5.5. In the same manner, changes to the class hierarchy i.e. existing classes or methods, can be assessed in comparing the redefinition profile for a single class obtained before and after modifications. Then, to capture an overview of the effects of changes, it is recommended to generate a redefinition profile for an isolated path or branch of the hierarchy.

In the previous experiments, the metrics results were either displayed in a tabular form or as bar charts. The graphical representation gave many insights on the redefinition mechanism and discovery of the MDR problem was possible. The bar chart graphical representation was expressive enough to suggest potential suspect defects and to reach satisfactory conclusions. However, in an interpretation process (section 3.4.3), other types of representations may be suitable depending on the subject assessed, the metrics used and the type of data obtained. The next section investigates several graphical representations for the metrics results. Then, a novel type of representation and its benefits are introduced in section 5.9.5. Then, section 5.10 shows how alarmers can be beneficial for the interpretation of specific phenomena on a metric profile.

### **5.9. Metric results visualisation and interpretation**

Large data sets are generally difficult to interpret. In the previous experiments, the use of the bar charts has contributed to the interpretation process. It is believed that the use of appropriate graphical representations facilitates the processing of the metrics results as well as the discovery of suspect features. Graphical representations permit a rapid depiction of phenomena occurring in the data set and depending on the data manipulated, a large variety (but not limited to) of standard graphic types is available and have various benefits. In addition, the combination of pre-processing functions on a data set prior to being visualised enables the detection of specific occurrences. For example, when only a portion of the data is desired, filtering functions can be used. In that respect, the purpose of this experiment is to evaluate a range of visualisations for supporting the interpretation process. In order to experiment with a variety of classical chart types, Microsoft® Excel97 was chosen as the graphical package application. The same data set i.e. the redefinition metric results obtained in previous sections, is used in order to keep elements of comparison consistent. In this experiment, the Smalltalk branches evaluated are the Object and the GraphicObject branches. These were chosen because they show completely different redefinition profiles and because potential design problems exist in the latter (see section 5.4.1). It is hypothesised that graphically displaying a data set using different representations may provide additional information for supporting the interpretation process. Therefore, the aim of this

experiment was to use different representations for the same data set in order to identify any interesting characteristics of each.

Note that explanations for the **Object** and **GraphicObject** bar charts were presented in section 5.6 and 5.4.1 respectively.

The key contributions of this section are:

- An investigation of various standard chart types in addition to a newly created one for the visualisation of the redefinition metrics results. The characteristics and benefits of each are explored.
- The concept of *alarmers* is presented and illustrates an example of application of pre-processing function on a data set.
- A data interpretation system is proposed for supporting the interpretation process.

### 5.9.1. Surface bar charts

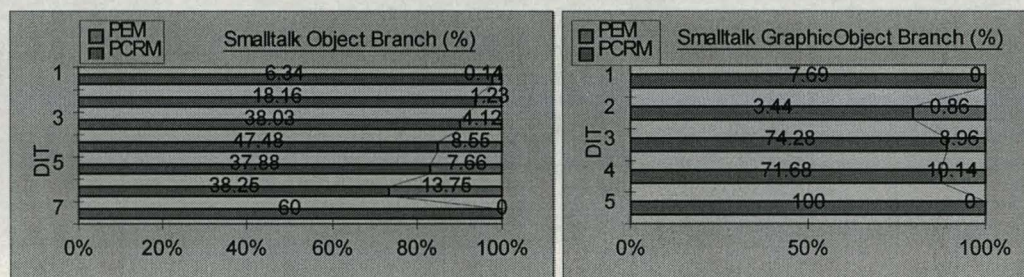


Figure 5.28 (a) and (b): Surface bar profiles for the **Object** and **GraphicObject** branches

Bar charts illustrate comparisons among measures in a data set, while surface bar charts combine the measures on the same percentage scale in such a way as to find optimum combinations between two sets of data, thereby highlighting any unbalanced distributions. The detection of such distributions is interesting for metrics such as the PCRM and PEM metrics (both variants of method redefinition). In Figure 5.28 (b), the general high proportion of PCRM compared to the PEM raises design questions regarding the use of the redefinition mechanism. For the **GraphicObject** branch, the extension of methods is poor. This visualisation is convenient for depicting trade-offs between metrics in a design where the design characteristics are anticipated. Notice that the join lines at the PCRM and the PEM boundary are drawn for ease of reading but do not define a smooth curve (the metrics results are discrete value sets). Further experiments on several other branches confirmed that the profiles shown occur on many occasions. An early analysis suggests two corresponding design problems:

- Methods in top classes are poorly abstracted. A 100% of PCRM for the **Object** branch at DIT=7 and for the **GraphicObject** branch at DIT=5 suggests a low level of polymorphic



methods in the top classes. Comparing Figure 5.28 (a) and (b) the visual effect of imbalance is immediate.

- Leaf classes are wrongly subclassed as they are not reusing inherited properties.

In Figure 5.28 (a), at DIT=6, the apportionment of PCRМ vs. PEM is 73.56 to 26.44% whereas at DIT=7, the apportion comes to respectively 100 to 0%. This suggests that leaf classes are more subject to complete redefinition than extension, however to discover the causes of such a situation, the analysis of the methods appearing at the concerned DIT is necessary. If further analysis of the measures depicted in the graphical representations is required, the behavioural inheritance analysis technique described in section 3.3.5 and used in chapter 5 is recommended.

### 5.9.2. Surface charts

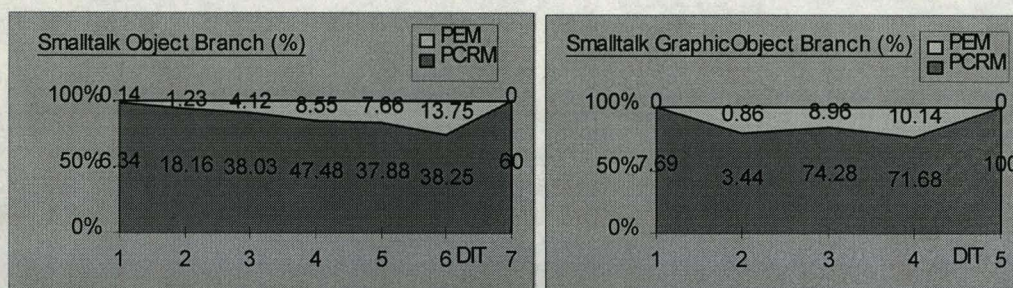


Figure 5.29 (a) and (b): Surface profiles for the Object and GraphicObject branches

The surface charts are used for the same purpose as the surface bar charts, however this representation is convenient for measures returning non-discrete values. On a scale of 0 to 100%, the representations of each proportion for each metric illustrate the disparities amongst the result set. In particular, it is possible to assess the magnitude of change of the measures over the DIT. This is intended only as an example<sup>27</sup> as the redefinition metrics return discrete values and is therefore unsuitable. Similarly to the surface bar charts, the surface charts quickly outline the balance between two or more correlated metrics.

<sup>27</sup> Notice that the x and y-axis have been interchanged for ease of reading.

## 5.9.3. Addition bar charts

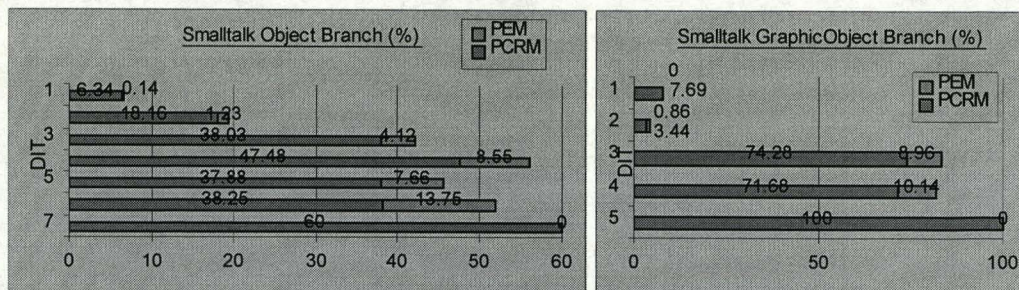


Figure 5.30 (a) and (b): Addition bar charts profiles for the Object and GraphicObject branches

The addition bar charts are a variant of the standard bar chart however, many measures can be “stacked” together on the same bar, thereby showing the relationship of individual measures to the whole. The contribution of each measure to the total is depicted. The addition bar charts are also suitable for complementary or related metrics. As completely redefined and extended methods are both considered as redefined methods, the sum of PCRM and PEM gives the PRM (Figure 5.30 (a) and (b)). In Figure 5.30 (a) and (b), PRM is shown by the total extent of the bar. The addition bar chart is considered an enhanced version of the simple bar chart as it makes clear the values for each of the shown metrics.

## 5.9.4. Radar charts

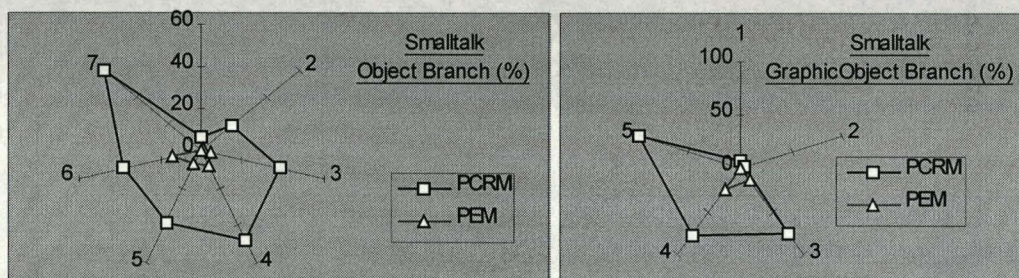


Figure 5.31 (a) and (b): Radar charts profiles for Object and GraphicObject branches

The radar charts allow the display of results across many dimensions. Each dimension has its own value axis radiating from the center point. The lines connect all measures in a particular data set. The radar charts permit rapid pinpointing of differences in the shape of the profile. In particular, it is convenient to use this representation when previous experiments have defined, for example, averages or thresholds for what is considered good or bad. Any disparity can then be depicted quickly. Again, the join lines are shown for ease of reading but it is possible to take them into consideration for identification of pattern profiles. When a smooth increasing curve is expected, the shape of the profile is a spiral. Attention should be taken when interpreting this type of chart as it can hold large amounts of data of different types e.g. different metrics across different DIT levels, that can clutter the graphic, and therefore the interpretation. For the GraphicObject branch, both curves obtained are rather intriguing as the redefinition activity seems to take place only in

deeper levels of the hierarchy. Intuitively, this is confirmed by the assumption that a class situated deeper in a hierarchy inherits all methods from its ancestor classes. It is therefore potentially able to call a high number of possibly unrelated methods, thus explaining the high level of redefinition. In Figure 5.31 (b), it is clearly seen that dimensions one and two are negligible compared to those remaining. Given that those dimensions represent the DIT level, it seems fair to conclude that a redefinition activity is more likely to happen in the bottom of the hierarchy and is due to the abstraction property of classes at the top. However, the rate of increase of the metrics cannot be easily pictured in those charts.

5.9.5. A colour coded range bar charts

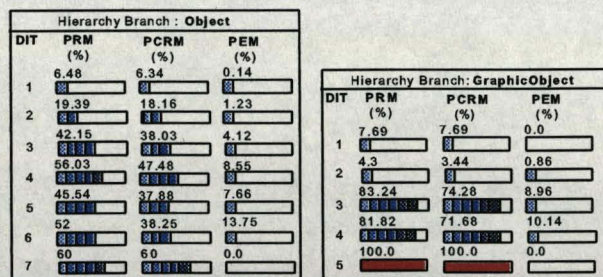


Figure 5.32 (a) and (b): Colour coded bar for the Object and GraphicObject branches

Range	Apportionment (%)	Colour coded bar
0	0	[ ]
1	0.01 - 14	[ ]
2	14.01 - 28	[ ]
3	28.01 - 42	[ ]
4	42.01 - 56	[ ]
5	56.01 - 70	[ ]
6	70.01 - 84	[ ]
7	84.01 - 100	[ ]

Table 5.2: Example of equally distributed ranges

In this thesis, the colour coded range bar charts have been created to address the issue of rapid threshold detection for metrics. These are adapted representations of the simple bar charts. In some cases, the display of ranges of values may be more relevant than the exact values for a particular data set. For example, metrics results can be compared to a range of thresholds rather than a single threshold value e.g. the 20% to 25% range. Instead of displaying the exact measures, the aim is to represent the ranges in which measures occur. To do so, the measures are pre-processed by a filter function. In addition, the use of colour for the different ranges gives extra information at first glance. The coloured bars shown in Figure 5.32 (a) and (b) have been obtained by checking the pre-defined ranges in which each metric value is situated. The coloured range bars are defined in Table 5.2. The apportionment has been arbitrarily chosen to be equal but this is not necessary. It is the responsibility of the designer to define the ranges and thereby the filter function, relative to

predefined threshold values. It is important to underline that this filtering method is not meant to be compared to a subjective assessment metric although it is based on the same principle as scaling. Table 5.2 shows an apportionment of a percentage scale into seven ranges, roughly equal to  $100/7$ . When the proportions are equal, the smaller the proportion is, the closer this visualisation will be to the equivalent in a bar chart representation. In the example, colour shaded rectangles have been used to give a gradual effect. It might also be interesting to consider non-equal apportionment of the ranges. In such cases, attention should be given to the grounds on which the proportions are attributed to prevent subjective interpretation [Hen96]. For example, adopting a non-equal range strategy for a metric  $m$  and, providing that previous statistical experiments deducted a threshold of 60%, only three ranges are necessary. The first range is for 0, the second from 0 to 0.6 and the third 0.61 to 1. The same principle of colour coded rectangles can be used to quickly locate defects, thus only three colours would be used in this example.

In the GraphicObject branch, from  $DIT=2$  to  $DIT=3$ , the peak (already pinpointed with the bar chart) appears even more suspect as the PRM increases by a factor of 21.6 suggesting potential design flaws at  $DIT=2$ . Although this visualisation seems similar to the bar charts, but less accurate, the main idea for such a visualisation is to use it in conjunction with a triggering function or alarmer.

#### 5.9.6. Visualisation uses

The different types of visualisation described in the previous sections support the metrics interpretation activity. It is believed that there is a need for integrating those visualisation techniques in a measurement programme. Further work is needed for identifying and extending the current recognised representations.

From the observations made on the experiments with the different visualisations, a summary table is given below in order to categorise and facilitate the choice of one or another. Each of the graphical representations is usually suited for a particular task i.e. pinpointing a particular characteristic of the data; therefore it is possible to categorise them depending on the purpose of the measurement and the task to be achieved. In the following table, for a particular task, the list of suitable visualisations and associated explanations is given.

Task	Visualisation	Explanation
Data evolution	<ul style="list-style-type: none"> <li>• Bar chart</li> <li>• Surface bar chart</li> <li>• Surface chart</li> </ul>	For the detection of peaks and general evolution of the data set. Also, identification of the localisation of the problems has been possible in the case study.
Correlation	<ul style="list-style-type: none"> <li>• Surface bar chart</li> <li>• Radar chart</li> </ul>	For the detection of disparate uses of an OO mechanism and trade off. It also permits the localisation of design problems with respect to related metrics. Often, the emphasis on the realisation of one of the criteria disfavours other criteria. This phenomenon is measurable and can be localised by defining the adequate metrics set.
Pattern profiles	Any charts with restrictions in the case of the alarmer	For the detection of possible repetitive pattern profiles corresponding to particular design problems in an OO system, the classification of typical profiles for later reference can be envisaged. This is currently being investigated in further work. A catalogue of typical good and bad profiles for a metric will be considered. Profiles from different branches are more likely to converge towards the same pattern as they employ the same object concept. Chidamber and Kemerer, in their empirical data collection, showed that the distribution of the results of their metrics converges even when the sites were different in terms of domain and OO programming language used [ChiKem94].
Alarmer	Colour coded range bar chart	For finding subset of data or single value within a given data set. The triggering mechanism of the alarm is defined by exact conditions.

Table 5.3: Summary of visualisation types

### 5.10. The concept of "alarms"

The concept of an alarmer is simple. Suppose we want to detect any factor increase  $> 2$  between two consecutive levels in the hierarchy. Any values satisfying the condition is expected to be pinpointed automatically. This is exactly what the alarmer technique is intended for. If an alarmer is set on for the GraphicObject branch in Figure 5.32 (b), only the values of PRM and PCRM at DIT=3 would be found. If it was decided to use the colour coded bar charts for visualisation, only the two bars at DIT=3 for PRM and PCRM are shown. Indeed, the visual effect of the colour

coded bar representation is immediate and asks for further analysis. The alarmer has accomplished its task in pinpointing the disparate results.

### The alarmer mechanism

The first desired functionality of an alarmer is that it should provide a means for defining the behaviour to be detected. A simple form of an alarmer would be to detect a particular expected value within a set. In such a case, a simple condition function would be sufficient to filter the initial results set. For instance, this would be useful for comparing metrics results to the traditional averages or threshold numbers. Suppose that after some statistical analysis of the redefinition metrics results for a project, a threshold of 40% of redefinition is arbitrarily defined above which the design is to be re-considered. Therefore the triggering condition is simply:

```
metricValue >= AVERAGE_THRESHOLD
```

The algorithm of such behaviour can be specified (example 1).

---

#### Example 1:

```
AVERAGE_THRESHOLD := 0.4.
```

```
SuspectedValues := Collection new.
```

```
(redefinitionAlarmer isOn)
```

```
  ifTrue: [
```

```
    metricResults do: [ :metricValue |
```

```
      ( metricValue >= AVERAGE_THRESHOLD )
```

```
        ifTrue: [
```

```
          suspectedValues add: metricValue.
```

```
          RaiseAlarm( metricValue ).
```

```
        ]
```

```
    ]
```

```
  ]
```

---

In the algorithm of example 1, the `AVERAGE_THRESHOLD` constant can easily be defined at run-time in an application. The `suspectedValues` collection contains the set of defect values. For this type of alarmer, a simple condition is sufficient to detect the desired characteristic i.e. `(metricValue >= AVERAGE_THRESHOLD)`. The `metricResults` is a collection of results values obtained from the derivation of a metric on a system. `metricValue` is a local instance variable equal to an item of the `metricResults` collection. The `raiseAlarm()` function can be a function which manages the presentation process of the alarm under a chosen form e.g. visual aspect or sound.

However, in the case of an alarmer triggered when the “weighted methods per class (WMC)” metric [Chidamber94] is greater or equal to 5, the triggering condition becomes a function:

```
wmc(class) >= AVERAGE_THRESHOLD
```

Then, the algorithm of such behaviour can be specified (example 2).

---

**Example 2:**

```

AVERAGE_THRESHOLD := 5.
SuspectedValues := Collection new.
(redefinitionAlarmer isOn)
  ifTrue: [
    systemToCheck do: [:class |
      ( wmc(class) >= AVERAGE_THRESHOLD )
        ifTrue: [
          suspectedValues add: class.
          RaiseAlarm( class ).
        ]
      ]
    ]
  ]

```

---

The difference in this example is that the triggering condition is now a function and not a single value. This condition is also tested for each of the classes contained in the `systemToCheck` collection of classes.

From the two examples cited, we can see that the core element of an alarmer resides in its triggering condition. In the case of large data sets, complex conditions can be applied. In a general case, an alarmer makes use of the following main components (Figure 5.33):

- **A filter function:** when not all metric values are of interest in the whole metric result set, a filter function can be used to reduce the amount of data processed.
- **A transformer function:** if the data has to be transformed before application of the triggering condition, a transformer function e.g. statistical functions can pre-process the metric results set.
- **A triggering condition:** defines the condition under which the set of values to check are satisfied.

### 5.11. Data interpretation system

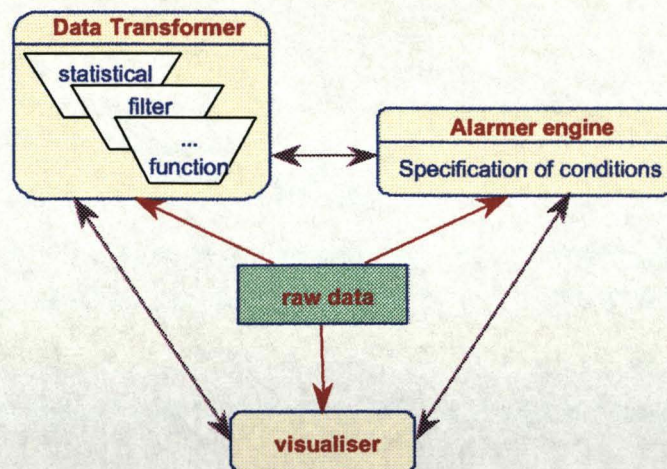


Figure 5.33: Data interpretation system

A data interpretation system has been built based on the components shown in Figure 5.33. The raw data in the model can be directly displayed or pre-processed before being displayed. The visualiser permits the display of the possible representations. A data transformer contains a list of functions permitting pre-processing of the data set. Typical transformer functions are filtering and statistical functions. When the designer has recognised some design problems in the hierarchy, the alarmer engine allows one to define and set up the alarm. In some cases, it is necessary to pre-process the data set before setting up an alarm for the new metrics set. Thus, the alarmer engine can co-operate with the data transformer.

The next section concludes the chapter on the experiments.

### 5.12. Conclusion of the experiments

Currently, one of the main problems that inhibits the development and adoption of OO metrics is a lack of tools for supporting their development and use in a general sense. Using the prototype developed, the experiments demonstrated that the redefinition metrics set is applicable to an object-oriented design, including designs not necessarily organised as a hierarchy. The metrics proved successful in the detection of suspect classes and thereby enabling the discovery of design problems such as the MDR problem. In addition, the graphical representations of the metrics results for various branches of the Smalltalk class hierarchy gave us insights into the behavioural aspect i.e. the method redefinition mechanism. The separation of the measures for the PCRM and PEM gave finer-grained indications on the ratios of redefinition at each level of the hierarchy.

In the context where the metrics generate large data sets, it is necessary to have some mechanisms to quickly filter or re-process the data set in order to facilitate their interpretation. The alarmer technique provides an easy way to detect problems that appears under certain conditions. If the triggering conditions are satisfied, the suspect values can be automatically pinpointed. The two



aspects of filtering and alarmer functions have been successfully demonstrated and the data interpretation system integrated within the prototype tool permitted the investigation of the colour coded range bar chart representation.

Due to the high-level of redefinition activity in some parts of the Smalltalk class hierarchy, it is possible to conclude that the inheritance mechanism is violated in many respects. To a major extent, the possible reasons behind such situations can be attributed to the weak type characteristic of the language. Also, in some cases, the lack of multiple inheritance clearly produces suspect design situations. Inheritance in current OO systems is still hazardous. A conceptual gap exists between OO modelling constructs and their mapping onto a language. The implementation of an inheritance relationship between classes using any OO programming language is actually a real source of design problems.

Chidamber and Kemerer's [ChiKem91, ChiKem94] early work on OO metrics proposed a suite of six metrics for assessing the complexity of an OO model. Their metrics were applied on C++ and Smalltalk. For each of their metrics, only simple histograms and summary statistics in a table form were produced. The interpretation of data relied on comparisons made between the histograms obtained for both sites. All charts represented the range of metric values (x-axis) obtained against the number of classes involved (y-axis) for each of the values. No dependency relationships between the metrics were presented. The authors only suggest that a class hierarchy can be "top" or "bottom-heavy" i.e. the DIT and the "*number of children* (NOC)" metrics are correlated. A high peak in the NOC histogram showed that most of the classes have no child classes. It was suggested that design practices dictated the use of shallow inheritance hierarchies, and that performance was the reason given in some cases. A use of surface bar charts might be a good candidate to exhibit previous observations. In such cases, it would be interesting to measure the number of classes per DIT level against their average number of children. Conceptually, it is expected the results would lead to the same conclusions.

In Lorenz and Kidd's [LorKid94] project experience database, only histogram charts were used. In some cases, this type does not seem appropriate due to the existence of large numbers in the results set. For instance, they considered the number of message sends metric and represented the values obtained against the number of methods. They correctly suggested that a rapid drop in numbers is the typical pattern found. This confirms the assumption that coupling between objects should be low in order to avoid inter-class dependencies. However, from a bad practice detection viewpoint, it would be more interesting to find out the methods which are strongly coupled. This could not be easily shown on the histogram provided as only a few methods are expected to send a large number of messages. Considering the colour coded range bar chart, an appropriate definition of ranges would immediately locate such peculiar results for further analysis.

An important area of measurement theory is the interpretation and analysis of metrics results. In our experiments, the analysis and interpretation process has been strongly supported by the method profiler feature of the prototype metric tool. In many cases, the precise location of suspect classes

containing methods with the MDR problem has been possible. At this point, it is possible to suggest that the MDR problem happens for at least three reasons:

- A class is wrongly subclassing its parent class i.e. the class does not satisfy the *is\_a* relationship.
- An incorrect design of interfaces of parent classes.
- A lack of abstraction of the top classes in the class hierarchy.

A possible solution for the first reason is to move the suspected class higher in the hierarchy so the class would inherit from the early implementation of the method, thereby minimising the chance for the MDR problem. In return, the class concerned will have to resolve all *super* calls to the original parent. This can be handled by the introduction of the original parent class as an aggregate which is instantiated in a constructor method. The great benefit of this solution is that it can be executed automatically. Otherwise, a manual intervention of the designer is probably required.

#### **Characteristics of the redefinition metrics**

The experimental validation of the metrics confirmed that the metrics measured the desired characteristics. However, concerning some abstract properties of good metrics mentioned by Kolewe [Kow93], alternative approaches are considered for the development of the necessary validation of the metrics. We shall briefly comment on these characteristics for the redefinition metric set:

- ✓ *noncoarseness*: we considered many different programs and were able to find different metrics results.
- ✓ *nonuniqueness*: if we consider two classes A and B derived from the same parent class where the same modifications on inherited methods are done and no added operations are made, we could find the PRMC is the same for both classes.
- ✓ *importance of implementation*: we assess a class's internal complexity by looking at its methods' redefinition. The metric depends on the implementation.
- ✗ *monotonicity*: not applicable for the redefinition metrics as their purpose is not to have a general value for the whole system. However, we could compute for two classes A and B their respective PRMC. Assuming that a class C contains all the methods from A and B with no name space conflicts,  $PRMH^C = PRMH^A + PRMH^B$ . For this characteristic, the redefinition metrics can be extended in order to calculate a mean value of redefined methods for a whole system.
- ✗ *nonequivalence of interaction*: same comment as previous characteristic.

- ✓ *interaction increases complexity*: as inheritance is a strong form of coupling and interaction is implemented via methods in a class, inheriting or adding new methods to a class increases its complexity, therefore the PRMH vary accordingly. Further verification requires to be done.
- \* *nonequivalence of permutation*: not applicable.

As the redefinition metrics are ratios that do not introduce arbitrary weightings or subjective values, the risk for wrong metrics' definitions is reduced. More importantly, the measures taken at each level of the hierarchy with the possibility of deriving the metrics on isolated branches permitted us to assess cross sections of an entire class hierarchy. This enabled a better understanding of the relevant abstractions in the hierarchy.

The next chapter concludes the research work and proposes a framework in which measurement techniques are "smoothly" integrated within an OO design process.

## **6. Discussion and conclusion**

*"Things should be made as simple as possible, but not any simpler." – Albert Einstein*

*"In general, no programming language or language mechanism should be used as a substitute for creative thinking, or as an excuse for avoiding software design and architecture." – Antero Taivalsaari*

The work presented in this thesis is concerned with the modelling issues of inheritance. It investigates the use of measurement techniques for the evaluation of goodness in an OO model. Ideally, the integration of metrics within the design activity is sought. Various aspects of inheritance in class hierarchies have been presented with a particular emphasis on the effects of the method redefinition mechanism. Based on the GQM process model, a measurement plan which lead to the creation of a novel redefinition metrics set (section 3.2) permitted the assessment of inheritance hierarchies. Analysis of the metrics results illustrated that the MDR problem (section 3.1.2) exists in class hierarchies. During the course of the measurement process, it was felt that the input of design considerations (section 3.2) was essential to the completion of the process. Experiments with the redefinition metrics were possible with the creation of a prototype metric collection tool for the Smalltalk class hierarchy. While the collection of the metric results have been possible, an appropriate analysis and interpretation of them proved difficult.

The main contributions of the work can be summarised as follows:

- In section, 3.1.2, the description of the multiple descendant redefinition problem in inheritance hierarchies. Different uses of the method redefinition mechanism showed that a model might violate the definition of inheritance although it may also satisfy the requirements. This reiterates the debate concerning the fundamental semantics given to the inheritance concept.
- In section 3.3, a description of design methodology considerations and techniques necessary for the assessment of inheritance, in particular the method redefinition mechanism. The design considerations describe an approach for identifying and gathering the information that is later utilised within the measurement process.
- In section 3.2, the definition of a set of candidate redefinition metrics for the assessment of use of method redefinition in class hierarchies.
- In section 3.4, a proposed metrics interpretation framework based on design methodology considerations and the method's life history analysis.

From a software engineering perspective, the designers benefit from the above contributions in many ways:

- Understanding of the causes and effects due to the presence of the MDR problem in class hierarchies. The use of the redefinition principle is still unclear. Papurt and LeJack [PapLeJ97] described the conditions under which method overriding should be used for three aspects: final, abstract and polymorphic. However, no consideration was given to the different types of redefinition. They consider method overriding as a replaced method, according to the classification given in section 2.2.4. Also, the authors mainly focused on the inheritance relationship between a parent and a child class but did not consider the life history of a particular method down a class hierarchy. The detection of MDR anomalies strongly suggests potential design problems that may compromise the future evolution of the design.
- The use of metrics gives insights into the improvement of the software architecture which is generally recognised as one of the key points of the design. Therefore, it also contributes towards the realisation of the requirements. In theory, an object model ought to be free from programming language considerations. In reality, as object-oriented languages offer a rich set of features, it would be unrealistic to completely ignore the implementation issues (see description of experiments in chapter 5). In consequence, these issues may directly affect the final design solution. As metrics are generally applied to the source code, all design issues can therefore be assessed. With the advent of modelling techniques using the concepts of components [Eng97], improvement of software architecture is made possible.
- One of the interesting aspects of measurement techniques (see section 2.4) is that they can be used as an instrument for problem discovery. The awareness and understanding of design problems enlightens the designers on the use of the fundamental object concepts. Recommended guidelines may be used during the whole design and assessment process.
- The use of measurement techniques not only improves the design solution but also contributes to the development of the design and measurement process. Further experiments are needed in this area in order to refine the technique and procedures involved in a measurement plan.

It is believed that the redefinition metrics and its variants are strong and simple candidates for detecting complex design problems occurring within a class hierarchy.

### **Metrics, method redefinition and implications**

Technically, the implementation of the redefinition mechanism is simple. Based on polymorphic selection or method body selection [PapLeJ97], different behaviour can be attached to the same method name and dynamic selection takes place at run-time depending on the object receiving the message, namely the execution of a method call. This mechanism gives code flexibility to the programmers. However, rather than a simple implementation exercise, the work presented in

chapter 2.4.6.1 emphasised the fact that the redefinition principle should also be regarded as a conceptual design tool. In our experience, most of the problems discovered concerning the use of method redefinition were design issues. To some extent, incremental design development and the mechanism of encapsulation are the two main reasons which increase the risk of incorrect redefinition use. For instance, if a designer is not the original author of an existing class hierarchy, careful attention should be given to the type of inheritance relationships used and the type of property modifiers for methods in classes. The behavioural inheritance scheme is not straightforward to understand especially if the hierarchy includes deep levels.

Only recently, CASE tools such as the RationalRose98® design tool support an automated visualisation of the inherited methods in class hierarchies. In addition to the methods defined by a class, it is also possible to visualise the list of methods inherited from the ancestor classes. Although this list does not include detailed information such as the origin of the method and the state of the method i.e. overridden or not, it is a valuable feature for the designers. Alternatively, in the recent Java documentation<sup>28</sup> format, a detailed textual description of the above is given. This partly fulfils the need for search mechanisms in class library documentation. It is clear that further modelling tools are needed to support the design tasks, in particular for class libraries. In the case of the method redefinition technique, a possible approach to verify the semantics of the inheritance relationships is to break down the tasks in two levels of abstraction. For each class, a systematic check is required for:

- **Immediate parent classes and subclasses:** in general, class hierarchies tend to be shallow rather than deep as recommended. Various types of inheritance contradict the conformance of classes in hierarchies. By consequence, classes tend to reuse behaviour from its closest parent classes rather than further classes. Thus, verifying that a class conforms to its nearest parent classes and repeating the process at all levels of the hierarchy guarantees that the inheritance relationship remains consistent.
- **Further parent classes and subclasses:** the previous level of abstraction permits a “localised” verification of the semantics of an inheritance relationship. In addition to this, an overview of the class hierarchy is also necessary because classes do not necessarily inherit their properties from their immediate superclasses. In the case of well abstracted hierarchies, it is common to encounter abstract methods in the root classes which are reused further down the hierarchy. Therefore, an overview of the resulting effect of encapsulation for a considered class is crucial.

The use of measurement techniques allows the detection of suspected design problems. When a problem has been identified, there are chances that an appropriate detection method can be found. In most cases, it is possible to find a pattern of code that corresponds to the design problem. Therefore, the identification of such patterns permits the discovery of the respective design

---

<sup>28</sup> Java development kit v1.2, Sun Microsystems, Inc. Copyright 1993-1999.

problems. For instance, “abnormal” super calls (section 3.3.4.1) may be detected with an appropriate lexical parser tool. Another example of inconsistencies is the MDR problem. Indeed, the method profiles obtained from the derivation of the redefinition metrics guided the search for the MDR anomaly. Also, the analysis of the method’s life history for multiple redefinition permits a localisation of potential suspect classes and methods. In many respects, measurement techniques represent an additional and valuable asset in the range of available design tools.

Taivalsaari [Tai98] stated that languages should not be a substitute for creative thinking. Therefore it is legitimate to consider their fundamental concepts and principles in the perspective of design assessment. Unfortunately, this situation does not encourage the important issue of separation of concerns between the design and the implementation phases. Similarly, it becomes tempting to tie design architecture issues to the supporting environment. This is not generally considered satisfactory.

Chidamber and Kemerer [ChiKem91, ChiKem94] proposed a suite of six metrics for assessing the complexity of an OO model. The DIT<sup>29</sup> metric is based on the following assumptions:

- A class situated deep in a hierarchy is more likely to inherit a great number of methods, hence increasing its complexity.
- A deep tree involves greater overall design complexity since the number of classes and methods are important.
- A class which is located deep in a hierarchy benefits from the potential reuse of inherited methods.

The redefinition metrics set adopts these assumptions; however, rather than using the DIT metric as a stand alone metric, it was incorporated it into the PRMH metric to give a more meaningful metric. The WMC metric is the weighted method per class which takes into account the static complexity of methods in a class. If the complexity is equal to one, WMC becomes simply the number of methods metric. Churcher and Shepperd [ChuShe95] showed that the metric was open to many interpretations when considering its use with constructors and destructors in C++. In addition, unlike the PRMH metric it makes no observations as to which methods are inherited and of those inherited, which are redefined and which are not.

Lorenz and Kidd [LorKid94] included in their metrics set the number of methods overridden by a subclass and produced an average extracted from tests on project results. However, unlike the redefinition metrics, it was done at class level only, no metrics were proposed at hierarchy level and system level. In addition, their metrics are not represented as percentages which clouds interpretation. For example, if number of overridden methods = 5, the class complexity is not

---

<sup>29</sup> The theoretical basis for the DIT metric came from Bunge's [Wan88] notion of the scope of properties.

the same if the class contains a total of 10 methods (50%) or if the class contains a total of 100 (5%).

The MOOD (Metrics for Object-Oriented Design) set [Bri&a195] addresses the evaluation of the main keypoints of mechanisms of the OO paradigm. The six metrics are: the method hiding factor (MHF), the attribute hiding factor (AHF), the method inheritance factor (MIF), the attribute inheritance factor (AIF), the polymorphism factor (PF) and the coupling factor (CF). MHF and AHF refer to encapsulation as they detect the amount of hidden attributes and methods. Again, no differentiation is made in the nature of the methods when deriving their metrics for inheritance. Thus, because of the possible existence of completely redefined methods within a class hierarchy, their measure of MIF and PF are affected and do not assess inheritance in such cases.

Lewis [Lew95a] proposed a set of fine-grained metrics for assessing overloading, overriding and polymorphism issues. Related metrics are the overridden method references (ORMR), the degree of method overriding (DMOR), the degree of polymorphism (DP) and the degree of obscured polymorphism (DOP). ORMR is applied at method or class level and is taken in the general sense of overriding. ORMR is aimed to be used with DMOR which counts the number of existing forms of a method in the whole application. DP relates to the justified use of method overriding but DOP seems to be language-dependent as it is directed at measuring unspecified polymorphic methods. None of their proposed metrics are considered as ratios and no case studies were presented.

Current research on OO metrics has not yet addressed the multiple descendant redefinition problem. The proposed metrics set was aimed at the assessment of a class hierarchy from a behavioural viewpoint and the detection of abuses of the method redefinition mechanism. The results shown in the experiments revealed that such abuses exist in the current Smalltalk Express hierarchy, but they are theoretically possible in any language. As suggested earlier this may be simply due to the inherent incremental development of a class hierarchy, especially when different people are involved in the development. It should be emphasised that a system can be in a perfect working state even when containing MDR anomalies. The MDR problem increases the code re-engineering difficulty and affects the natural extension of the inheritance tree which degenerates in the presence of MDR (see section 3.1.3).

To support the interpretation of results obtained from the redefinition metrics, additional tools were required to precisely pinpoint defects in methods. The method profiler realised that task by providing a life history for each redefined method of each class along a particular branch of the hierarchy. The analysis of suspect classes was facilitated. A possible approach to further refine the redefinition metric set is to detect complex redefinition cases described in section 3.3.4.1. Although this would provide detailed information about the behavioural aspect, it pre-supposes that the metric would become language dependent. Again, it can be argued that such complex redefinition cases can be considered as design or implementation issues. Further work is needed in this area.



### Metrics collector tools

Despite the fact that the simple functionalities of the metric tool were enough to demonstrate the applicability of the redefinition metrics, it is possible to identify a number of future development areas as follows:

- The tool requires an appropriate versioning system for storing measures on the same subject at different points in time. This would be particularly beneficial for enabling comparisons on designs that continuously evolve with time. The current solution adopted is to save the method profiles as textual files, delete the profile from the persistent repository and finally to re-calculate the metrics when necessary. Indeed, the textual files contain the metric results and therefore are available for further processing tasks.
- In its current state, the metric collector tool lacks automatic transfer of metric results to a graphical tool such as Microsoft Excel®. In the experiments, manual copies of the result values were necessary in order to be processed. A possible solution is to use the *Object Linking and Embedding* mechanism provided by the Microsoft Windows™ environment. However, as a possible future development, it is desirable to extend the current functionalities for the management and analysis of the metrics results. For instance, the graphical representations could be done within the same package and further re-processing algorithms of the metrics results can be developed.
- The development of a metrics' definitions repository is crucial for the extension of the prototype tool. As proposed in [SimLew98], the work constitutes an entire topic of research on itself. Similarly, further investigations for a common architecture towards a flexible structure for metrics repositories are desired.

In conclusion, the metric prototype tool successfully demonstrated that the redefinition metrics is applicable and that automatic collection of measures is possible. A simple tabular display of the metric results gave insights on the method redefinition profile of the Smalltalk class hierarchy. Given the simplicity of the architecture, it was shown that the development of such a tool is facilitated by the presence of functionalities to extract meta-information. The last of the points mentioned above showed the need for an improved version of the architecture of the persistent repository. This confirms the fact that the use of a metric tool collector alone is not enough and requires support from other tools. It should be emphasised that the discovery of unexpected use of inheritance was possible when collecting the measures on branches of the Smalltalk hierarchies. Further investigations and development were needed for the interpretation of the metric results. In its current state, the metric tool satisfied the original requirements but could be extended for further functionalities.

### **GQM lacks the pre-assessment and the interpretation phases**

Although it seems natural to know what to measure before measuring, the identification of the appropriate attributes in relation to the purpose of measurement is difficult to establish. Similarly, past experiments with metrics [Fen90, Fug&al98, HarNit96, Hen96] clearly illustrate the problematic issues in interpreting metric results. There is always the risk that correct metric results may suggest incorrect conclusions or unwanted actions. The problem of interpretation concerns the techniques or approaches taken for deducing conclusions. For example, the use of arbitrary thresholds essentially infers three categories of conclusion: the results may be greater, lower or equal to the threshold. This technique assumes that the comparison with such a value is possible. However, the interpretation task requires the knowledge of the context of measurement. Values under a threshold on a curve may not necessarily indicate normality. In the experiments with the hierarchy redefinition metrics, for a particular level in the hierarchy, an "abnormal" PCRM value for a class may be leveraged, therefore hidden, by the low PCRM values in other classes. Thus, a thorough analysis of metrics results obtained together with input from the design task enable the designers to confirm or refute their initial hypothesis, and thereby take appropriate action.

In [Fug&al98], the authors describe their experiences in applying the GQM approach in industry. In addition to the identification of drawbacks in the use of GQM, interesting recommendations and suggestions were given concerning the specialisation of the approach within a large software house. It is particularly striking how the authors emphasised the needs to understand the company business rules before establishing the list of goals for the assessment plan. This was necessary in order to effectively customise the GQM plan to the company and to avoid unrealistic goals. In the following example, the basic format of the goal definition is shown:

**Analyse the introduction of GQM measurement technology  
for the purpose of better understanding  
with respect to cost/benefit ratio  
from the viewpoint(s) of the quality organisation and project team  
in the following context: experimental sites of the CEMP project**

Although their measurement plan mainly concerned the process level, analogies can be drawn with the work in this thesis where a pre-assessment phase was required prior to the use of product metrics. In order to define the correct goals and metrics, it is essential to have concise ideas about the application requirements and the attributes assessed. From the experiments, it is clear that the assessment of object-oriented models would not be as beneficial without a good understanding of the design process and the experience gained from previous design exercises. The assessment of inheritance hierarchies was driven by the aim of discovering unexpected inconsistencies in the presence of method redefinition. Given the knowledge of possible interpretations of the inheritance model, it was possible to focus on the method redefinition technique for a behavioural inheritance analysis.

In [Fug&al98], an improvement of the GQM process has been realised by inserting an additional step i.e. *the abstraction sheet*, which aims at bridging the gap between the goals and metrics definition stages. To this end, for each goal, the focus, variation factors, hypotheses and the expected impact of variation factors on the hypotheses are summarised. Abstraction sheets were found useful in capturing the implicit knowledge about the process or product. Both the method's life history analysis and the additional abstraction sheets step in the GQM process illustrate how experimental approaches permit a refinement of the measurement process itself.

The experiments with the redefinition metrics gave us insights into their practical use. Undoubtedly, the use of metrics is not straightforward as many technical issues are involved in the process. In fact, it is clear that this process currently lacks design considerations that are geared towards the definition of an assessment programme. For instance, the redefinition profiles obtained from different branches of the Smalltalk class hierarchy permitted the identification of possible pattern profiles regarding the category of classes assessed. However, the interpretation of the metric results would not be realistic without referring back to the design problem. To date, no interpretation methods exist concerning the analysis of the property inheritance scheme with the use of metrics. Pragmatically, it is possible to draw an example list of aspects to review during the interpretation process:

- The goals of measurement.
- Identification of potential design problems and hidden side effects should be possible.
- Any possible mismatch between the requirements, the detailed design specifications and the implemented solution.
- The object oriented concepts involved and their multiple interpretations.
- The assumptions made on the design and during measurement.
- The designers' point of view.

Ideally, the designers ought to discover the reasons behind the phenomena shown by the redefinition profiles. Then, a relation from cause to effect can be established between observed phenomena, the generated design problem, the context in which the problem occurs and the possible directions for improvement. In addition, with the new findings, a refinement of the measurement plan and the metric set can be made.

A proposed additional refinement step as a new stage in the GQM process can be as follows: *the analysis and interpretation step* as described in section 3.4.3 appears to be a natural step which takes place after the metrics definition stage in the GQM process. The interpretation framework is composed primarily of the three aspects: the raw data representation, the profile analysis and the designer's feedback. The framework is intended to describe to the different aspects to be reviewed during the interpretation phase. Although the emphasis was given to graphical representations, it

was not intended to cover all possible visualisation techniques. This requires further work and represents a separate topic of research.

### **Further work**

In order to complement the work presented in this thesis, a number of immediate areas can be identified as follows:

- The investigation of effects of Java interface mechanism on the use of method redefinition.
- The creation of new types of representations for the results of design metrics for OO systems.
- The classification of typical pattern profiles. Further tests are needed to explore the possibility of defining good or bad pattern profiles for these metrics. This new area of research seems to be promising and should be considered as a part of the software measurement process as well as the software development life cycle.
- The formalism of specification of the triggering condition for the alarmer.

It is believed that visualisation techniques and the concept of alarmers for data interpretation provide a more expressive approach to interpreting metric results thereby enable the detection of complex design problems.

From a broader perspective of the project, there is a need for an integrated development environment whereby measurement techniques are used to assess an OO model at early stage of the development and also to be able to re-inject design decisions into the model. Thus, the design-evaluation cycle can be completed and repeated. In summary, "*measure to understand, interpret to decide and transform to improve*". The final section of this thesis opens the way for such integration. Naturally, the proposed data interpretation model can be seen as part of a measurement framework model such as "*the application of metrics to industry (AMI)*" program proposed in [Row93].

### **On the integration of measurement techniques in an object-oriented design process**

In light of the work presented in this thesis, it is proposed that measurement techniques and the process of object-oriented design should be considered part of the same development process and not act as two different tasks as currently is. This thesis cannot cover all necessary aspects involved in such desired integration. However, having concentrated on one possible use of metrics to assess inheritance, it is possible to suggest directions for improvement of the current design process. The main problems encountered during the experiments with metrics were the lack of similar results from other experiments for comparison. To palliate this deficiency, a refocus on the goals' definition and the analysis of methods' life history supported by the interpretation framework enabled satisfactory conclusions on the experiments. This emphasises the fact that a

good understanding of the design is necessary before the start of a measurement programme [HenEdw94, Whi97]. The new interpretation model which is part of the measurement process and presented in section 3.4 directly addresses this deficiency. This work has demonstrated that metrics are beneficial in many respects; nonetheless, it should be noted that feeding results back into the design remains difficult due to the necessary effort for potentially re-designing and re-engineering the code. This would imply additional cost on the overall development; thus the relative “unpopularity” of the measurement science amongst the software engineering community. It is believed that such situations can be smoothly tackled in adopting an iterative and incremental development approach.

Figure 6.1 depicts an overview of the current situation concerning the interactions between the modelling tasks and the assessment tasks. Three different layers: the “Requirements”, the “Processes” and the “Deliverables” are represented for the purpose of identifying the interactions between the modelling and assessment tasks. Due to the relatively recent interest of researchers in assessment techniques, a clear separation between the two processes exists. Rather than being integrated at process-level, the assessment activities are co-ordinated at the deliverable level. The progress of the assessment methods depends on the state of the outcome from the design methods i.e. the OO model or the source code.

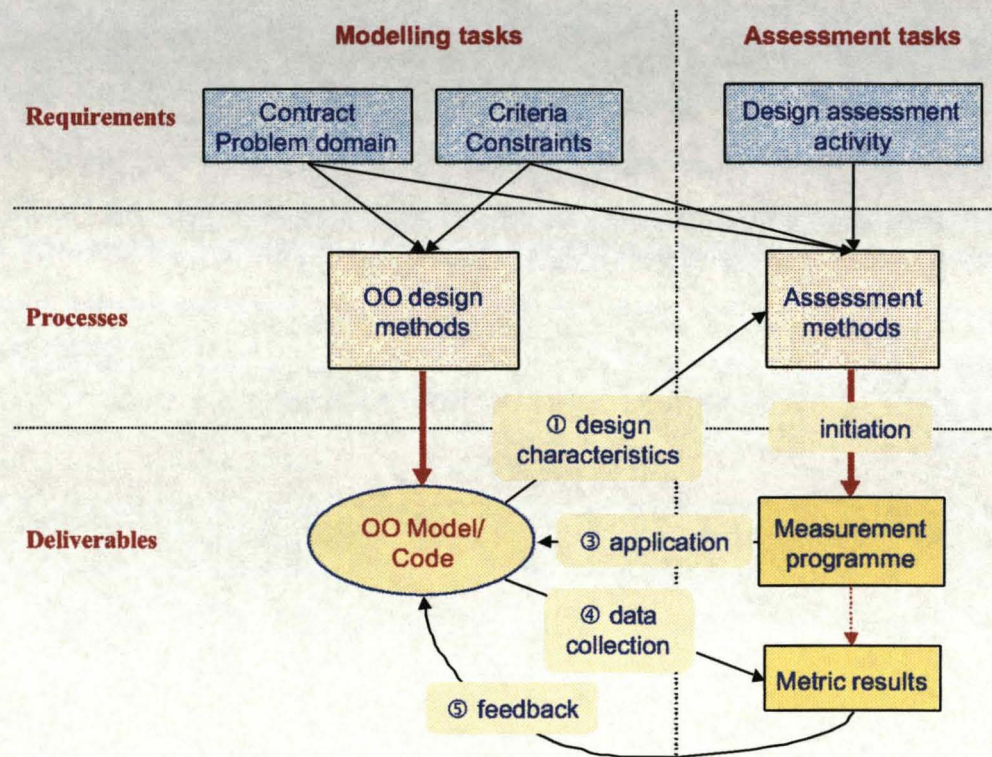


Figure 6.1: Modelling and assessment tasks

In Figure 6.1, a simplified sequence of assessment activities is given by the numbered labelled arrows:

1. Design characteristics gathering: when the assessment activities start, an informal survey of the design is done in order to identify the goals of measurement.
2. Measurement programme initiation: the programme is defined and suitable metrics are identified.
3. The derivation phase corresponds to the application of the metrics on the subject attributes.
4. Data collection.
5. The feedback phase is expected once the metrics results are analysed.

Further work is necessary on the identification of core product metrics for the use of object concepts. The experiments show that a merging of assessment activities and design is necessary in order to complete the measurement programme. In particular, for maximising the chances for better decision making from the analysis and interpretation phases, the designer must rely on previous design decisions. Some example benefits include:

- Until now, OO design methods do not include any form of evaluation method, thus risking a mismatch between the requirements and the developed application. Assessment techniques are one potential candidate for filling this gap. An integrated model would promote the inclusion of measurement concerns within the design activities. Systematically assessing a candidate object model has as a first objective the demonstration that the object concepts are correctly utilised and secondly that the necessary abstractions and behaviour are adequate to the requirements. In a different perspective, the choice of the best-suited design amongst a set of possible candidates may be possible with the use of metrics i.e. quantification of level of goodness.
- While a measurement plan at design phase may involve additional costs on the overall development, design problems may be even more expensive to rectify in the future. Unfortunately, tight development budgets often imply that software development is reduced to the simple phase of implementation where all the design decisions are made without a real overview of the essential architectural issues. In consequence, it is not rare to observe that, in many cases, the complete redevelopment of the software is necessary when new requirements appear.
- At the current state of research, existing measurement models are flexible and open enough to be integrated within the design process. In fact, measurement techniques naturally fit into an incremental development process as illustrated in Figure 6.2.

### **Proposed integrated model**

Besides the human and the organisational aspects, the proposed integrated model mainly aims at providing designers with a simple framework which co-ordinates assessment activities with design. In all cases, such integration ought to be as smooth as possible in the sense that it should not disturb or deviate from the design goals. Here, the term “integration” refers to a high-level integration rather than a detailed description of the model. An OO design process describes a set of activities, partially ordered and potentially dependent on each other. A measure is a quantitative element related to the presence of a specific attribute in the object model. Therefore, a possible approach for integration consists of identifying at what stage of the design process a targeted attribute appears in the model. Then it would be possible to derive the metrics. However, further conditions are required before being able to do so. A possible situation where potential wrong measures can be taken is when the attributes assessed are not in a consistent state. Recall that one aspect of the design is that it evolves constantly until its final version. As it is during the course of design that the benefits of the measurement techniques are desired, the start of such a programme will depend on the state of the object model. Therefore, the identification of the “critical” design activities, i.e. activities that enable the model to reach a correct and working state, determine if a measurement plan is possible. In such a case, a guideline may be defined as follows:

---

#### **Measurement guideline:**

The use of measurement techniques during the design process may only be envisaged if:

- All various forms of the abstractions or attributes targeted are identified. Note that the candidate metrics should only address one particular form of an attribute at a time.
  - The identified abstractions or attributes are stable. The stability of an abstraction or an attribute relates to their correctness during the design phase. An essential condition is that the candidate object model satisfies the requirements, therefore providing a consistent stable design point.
  - The design activities that produce the abstractions or attributes are known. The recognition of these activities may be not straightforward as the design process itself is not necessarily a strict sequence of the same activities. However, when the abstractions or attributes are recognised to be stable, the identification of desired activities is possible.
- 

Given that the process of design is constantly evolving and that the measurement techniques can be applied at stable design points, a “natural” integration of both activities is possible in the perspective of incremental development. Here, the term integration can be defined as a co-operation between the two activities based on the exchange of inputs and outputs. Figure 6.2 depicts the integration of an incremental design process with a modified version of the GQM plan.

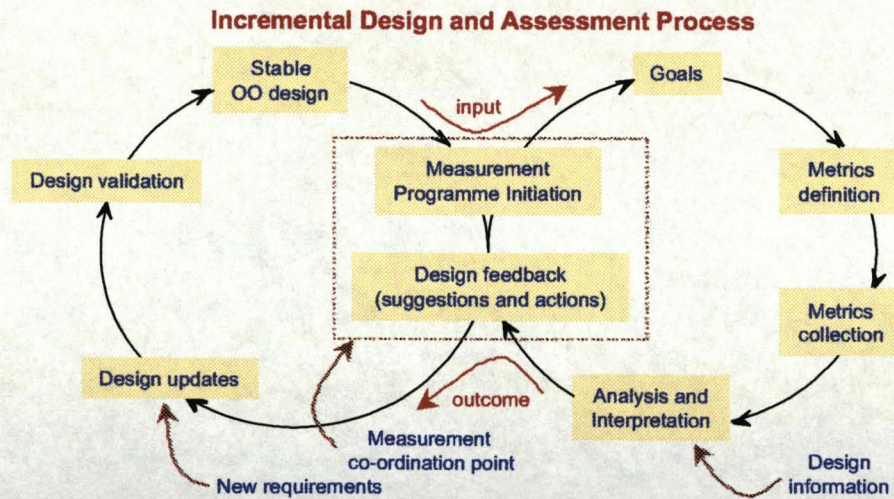


Figure 6.2: Incremental Design and Assessment Process

An incremental design process is represented as a simple loop sequence of requirement inputs, design updates and design validation. The measurement plan shown in Figure 6.2 is based on the GQM plan [Bas&al94] and mainly includes the findings from section 3.4.3 for the analysis and interpretation phase. In consequence, a possible smooth integration simply consists of the insertion of measurement plans at all identified stable design points in the design process. Thus, a stable point of the design determines the start of the measurement initiation phase. The point of integration between design and measurement activities is referred to as the *measurement co-ordination point*. Basically, the object model produced at this stage becomes the input for the measurement programme. In return, design feedback is expected as outcome from the analysis and interpretation phase. As a consequence, the design improvement suggestions serve as inputs, as well as any new requirements for a new phase of the incremental design. Recall that the designer's intervention is crucial for a correct analysis and interpretation of the metric results. This is characterised by the design information input in Figure 6.2. Figure 6.3 shows an overview of the proposed integrated model.

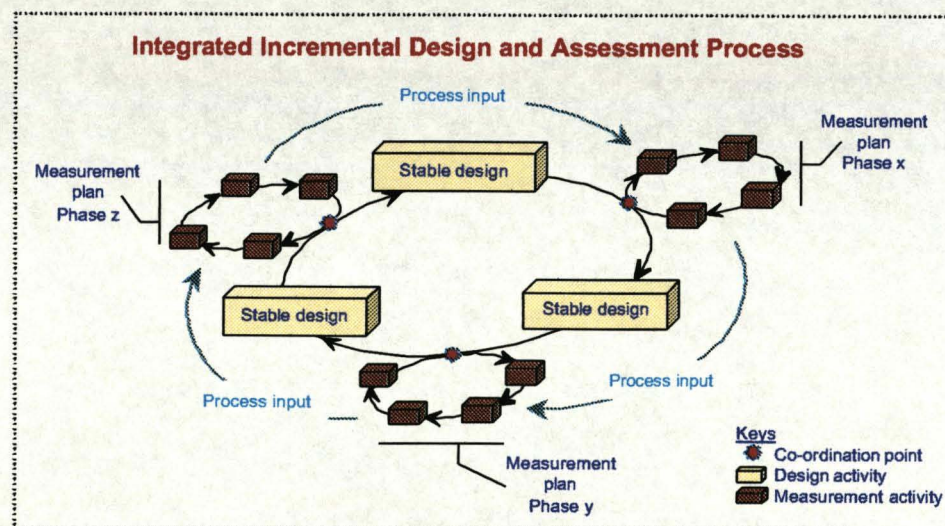


Figure 6.3: Integrated model for design and assessment



A simplified example of the incremental design process is pictured in the centre of the diagram with only three stable points. Each of these boxes hides all the design activities necessary to reach a stable point. Pictured as separate processes on Figure 6.3 between each of the stable points, a measurement plan is grafted on the core design cycle. Inasmuch as the design requirements and issues are being tackled, the important aspect of such integration is that the measurement processes themselves evolve. Two consecutive measurement plans may not be related depending on the attributes assessed. If it is the case, a review of the previous plan is necessary to take into account any new design information. Although the goals may remain the same, the corresponding attributes to be assessed may have changed due to the changes occurring in the object model. Sometimes, for the same problem, a different design solution is adopted from one stable design to the other.

The main beneficial aspect of this framework is that it keeps both processes separate and independent while co-ordination and co-operation are possible. The model remains flexible and no constraints are imposed on the design activities. The measurement co-ordination points are the input and output exchange from the design to the measurement tasks and vice-versa. Details of the related design information can be found in section 3.2. In many respects, such a model was unconsciously applied during the course of the experiments in this research work.

The above description of a proposed integrated model of measurement techniques within an object-oriented design process give us directions for challenging and interesting future work. Although the general description of the model has been given, further issues have to be tackled regarding the definition of a concise methodology. For instance, it is believed that profiles such as the redefinition profile (chapter 5) correspond to particular design situations e.g. MDR problem. A dictionary of such profiles, in particular for identified design problems, would prove beneficial for the designers. In the same manner as with a medical doctor, the identification of symptoms would suggest the causes and effects of the problems. Another promising area of research concerns the dependencies between metrics (section 2.4.6.1). From a re-engineering perspective, these dependencies are the key for enabling proactive design feedback from the use of metrics. If the dependencies were quantitatively defined then it would be possible to predict how the metric values vary if one or another varies. Therefore, such a technique can act as a simulation instrument for inferring the corresponding future evolutions of the current object model.

Perhaps the inheritance mechanism itself still deserves more attention since no agreement exists on the diversity of its application. Clearly, it is the understanding of business problems that drives the design of languages and therefore, the architecture of the design. For example, the Java language encompasses such a comprehensive set of class and method modifiers that their combination with other aspects of the design make it difficult to master. Architectural issues are probably the essence of the design process and further development of appropriate metrics is also needed.

In conclusion, this thesis presented an illustration of the potential benefits of measurement techniques regarding the complexities of the concept of inheritance. For years, it has been generally accepted that measurement techniques are mature enough to take part in an industrial process. Unfortunately, the reality is still otherwise and experiments within industry are rare. Perhaps the main causes of such a situation relates to the rapid evolution of OO concepts for designing, and the progress of programming languages and other associated technologies for solving enterprise business problems.

The notion of compromise or trade-off remains the key element in the decision process. However, all factors influencing the compromise must be known. The complexity of applications and the development process require the contribution of various resources from designers, abstraction tools and methods. A design assessment framework is one possible solution to ensure the success of each of the design milestones. It is a natural desire to evaluate goodness, originality and creativity in object-oriented design. Assessment techniques contribute strongly to this goal, so let's design and measure, and vice versa!

## References

[Abb83]

R. J. Abbott, "Program Design by Informal English Descriptions", *Communications of the ACM*, Vol. 26, No. 11, pp. 882-94, 1983.

[Abb&al94]

David H. Abbot, Timothy D. Korson and John D. McGregor, "A Proposed Design Complexity Metric for Object-Oriented Development", Department of Computer Science, Clemson University, Clemson, SC29634-1906, 1994. <http://www.cs.clemson.edu>.

[AdaMol95]

Jim Adamczyk and Tom Moldauer, "Trading Off: Inheritance vs. Reuse", *Object Magazine*, pp. 56-59, September 1995.

[AksBer92]

Mehmet Aksit and Lodewijk Bergmans, "Obstacles in Object-Oriented Software Development", *OOPSLA '92 Conference proceedings*, Vol. 27 of ACM SIGPLAN Notices, pp. 341-358, New York, October 1992.

[AlAsp93]

A. Al-Janabi and E. Aspinwall, "An Evaluation of Software Design Using the DEMETER Tool", *Software Engineering Journal*, pp. 319-324, November 1993.

[Avo94]

Jon Avotins, "Defining and Designing a Quality OO Metrics Suite", Technology of Object-Oriented Languages and Systems (TOOLS) Conference proceedings, USA, 1994.

[ArmMit94]

James M. Armstrong and Richard J. Mitchell, "Uses and abuses of inheritance", *Software Engineering Journal*, January 1994.

[Bak&al90]

Albert L. Baker, James M. Bieman, Norman Fenton, David A. Gustafson, Austin Melton and Robin Whitty, "A Philosophy for Software Measurement", *The Journal of Systems and Software*, No. 12, pp. 277-281, 1990.

[Ban97]

Jagdish Bansiya, "A Hierarchical Model for Quality Assessment of Object-Oriented Designs", *Ph.D. Thesis*, University of Alabama, Huntsville, Alabama, , October 1997.

[BarSwi93]

G. Michael Barnes and Bradley R Swin, "Inheriting software metrics", *Journal of Object-Oriented Programming*, pp. 27-34, Nov/Dec. 1993.

[Bar&al93]

Patrick Barril, Mahmoud Boufaida and Jean-Francois Brette, "Class Cooperation in a dedicated Object System: the FORCE Authoring Environment", *TOOLS Europe '93 Conference proceedings*, pp. 115-123, 1993.

[Bar&al97]

Mario R. Barbacci, Mark H. Klein and Charles B. Weinstock, "Principles for Evaluating the Quality Attributes of a Software Architecture", *Technical Report CMU/SEI-96-TR-036, ESC-TR-96-136*, May 1997.

[Bas&al94]

Victor R. Basili, Gianluigi Caldiera and H.Dieter Rombach "The Goal Question Metric Approach", *Encyclopaedia of Software Engineering*, John J. Marciniak, Vol. 1, pp. 469-476, John Wiley & Sons, 1994.

[Bas&al95]

Victor R. Basili, Lionel Briand and Walcelio L. Melo. "A Validation of Object-Oriented Design Metrics as Quality Indicators", University of Maryland Institute for Advanced Computer Studies, Dept. of Computer Science, Univ. of Maryland, Technical Report CS-TR-3443, May 1995.

[BanKim87]

Jay Banarjee and Won Kim, "Semantics and Implementation of Schema Evolution in Object-Oriented Databases", ACM 0-89791-236-5, pp. 311-323, 1987.

[BecGua92]

Lee Becker and Todd Guay, "Object-Oriented Diagnosis", *Journal of Object-Oriented Programming*, pp. 43-52, October 1992.

[BecJoh94]

Kent Beck and Ralph Johnson, "Patterns Generate Architectures", *ECOOP '94 Conference proceedings*, July 1994.

[BecGua92]

Lee Becker and Todd Guay, "Object-Oriented Diagnosis", *Journal of Object-Oriented Programming*, pp. 43-52, October 1992.

[Ber91]

Paul L. Bergstein, "Object-Preserving Class Transformations", *SIGPLAN Notices*, ACM Press, Vol. 26, No. 11, pp. 299-313, Phoenix, AZ, November 1991.

[BinSch96]

Aaron B. Binkley and Stephen R. Schach, "Impediments to the Effective Use of Metrics Within the Object-Oriented Paradigm", *.OOPSLA '96 Conference proceedings*, Workshop on "OO Product Metrics", Workshop report, 1996.

[Bis97]

*The BISS AWT Framework*, BISS GmbH, <http://www.biss-net.com/doc/biss-awt.html>, 1997.

[Bla92]

Michel Blaha, "Models of Models", *Journal of Object-Oriented Programming*, Vol. 5, No. 5, pp. 13-18, September 1992.

[Bla93]

Michel Blaha, "Aggregation of Parts of Parts of Parts", *Journal of Object-Oriented Programming*, pp. 14-20, September 1993.

[Boh88]

Bohem, B., "A Spiral Model of Software Development and Enhancement", *IEEE Computer*, March 1988.

[Boo91]

Grady Booch, *Object oriented design with applications*, Benjamin/Cummings, 1991.

[Boo94]

Grady Booch, *Object-oriented analysis and design with applications*, Benjamin/Cummings, 1994.

- [Bou89]  
Barbara M. Bouldin, "What Are You Measuring? Why Are You Measuring It?", *Software Magazine*, Vol. 9, No. 10, pp. 30-39, August 1989.
- [BraCoo90]  
Gilad Bracha and William Cook, "Mixin-Based Inheritance", *OOPSLA/ECOOP '90 conference proceedings*, Canada, 1990.
- [Bre96]  
Rolf Breuning, "Concept and Implementation of Multiple Inheritance in Smalltalk", *TOOLS Europe '96 Conference proceedings*, pp. 185-194, 1996.
- [Bri96]  
Sharmila R. Bristol, "Tools for Object-Oriented Metrics Collection", *OOPSLA '96 Conference proceedings*, Workshop on OO Product Metrics, 1996.
- [BriCuc98]  
Fernando Brito e Abreu and Jean Sébastien Cuche, "Collecting and Analyzing the MOOD2 Metrics", *ECOOP '98, Object-Oriented Product Metrics for Software Quality Assessment Workshop*, Brussels, Belgium, July 21, 1998.
- [Bri&al95]  
Fernando Brito e Abreu, Miguel Goulão and Rita Esteves, "Towards the Design Quality Evaluation of Object-Oriented Software Systems", *Proceedings of the 5th International Conference on Software Quality*, Austin, Texas, USA, October 1995.
- [Bri&al94]  
Lionel Briand, Sandro Morasca and Victor R. Basili, "Goal-Driven Definition of Product Metrics Based on Properties", University of Maryland Institute for Advanced Computer Studies, Dept. of Computer Science, Univ. of Maryland, Technical Report CS-TR-3346, September 1994
- [Cas93]  
Eduardo Casais, "The Automatic Reorganization of Object Oriented Hierarchies", *Building Object Oriented Software Libraries*, Eduardo Casais and Claus Lewerentz (Eds.), FZI-Publication 6/93, Forschungszentrum Informatik Karlsruhe, 1993.
- [CapLee93]  
L. F. Capretz and P. A. Lee, "Object-Oriented Design: Guidelines and Techniques", *Information and Software Technology*, Vol. 35 No. 4, pp. 195-206, April 1993.
- [Car98]  
Michelle Cartwright, "An Empirical View of Inheritance", *Information and Software Technology*, Vol. 40 No. 14, pp. 795-799, ISSN 0950-5849, 1998.
- [Cha97]  
Craig Chambers, *The Cecil Language, Specification and Rationale, version 2.1*, Dpmt. Of Computer Science and Engineering, University of Washington, March 1997.
- [ChaFau92]  
Dennis de Champeaux and Penelope Faure, "A Comparative Study of Object-Oriented Analysis Methods", *Journal of Object-Oriented Programming*, Mar/Apr 1992.
- [Cha&al92]  
Dennis de Champeaux, Doug Lea and Penelope Faure, "The Process of Object-Oriented Design", *OOPSLA '92 Conference proceedings*, ACM SIGPLAN Notices, Vol. 27, No. 10, pp. 45-62, October 1992.

- [Che81]  
Chen, Peter P., "Entity-relationship approach to information modelling and analysis", *Proceedings of the Second International Conference on Entity-Relationship Approach*, Washington, D.C., October 12-14, 1981.
- [CheBel94]  
Cheswick W.R., Bellovin S.M., *Firewalls and Internet Security*, ISBN 0-201-63357-4, Addison-Wesley, 1994.
- [CheHua93]  
Deng-Jyi Chen and Shih-Kun Huang, "Interface for Reusable Software Components", *Journal of Object-Oriented Programming*, pp. 42-53, January 1993.
- [CheHun94]  
Jen-Yen Chen and Yu-Shiang Hung, "An Integrated Object-Oriented Analysis and Design Method Emphasizing Entity/Class Relationship and Operation Finding", *Journal of Systems and Software*, Vol. 24, pp. 31-47, 1994.
- [CheLee95]  
Jan-Bon Chen and Samuel C. Lee, "Pursuing safe polymorphism in OOP", *Journal of Object-Oriented Programming*, pp. 39-45, March-April 1995.
- [CheLee96a]  
Jan-Bon Chen and Samuel C. Lee, "Generation and Reorganization of Subtype Hierarchies", *Journal of Object-Oriented Programming*, pp. 26-35, January 1996.
- [CheLee96]  
Jan-Bon Chen and Samuel C. Lee, "The necessary and sufficient conditions of type-safe polymorphism", *Journal of Object-Oriented Programming*, pp. 33-42, February 1996.
- [CheLu93]  
J-Y Chen and J-F Lu, "A New Metric for Object-Oriented Design", *Information and Software Technology Journal*, Vol. 35, No. 4, pp. 232-240, April 1993.
- [ChiKem91]  
Shyam R. Chidamber and Chris F. Kemerer, "Towards a Metric Suite for Object-Oriented Design", *Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'91) Conference proceedings*, pp. 197-211, October 1991.
- [ChiKem94]  
Shyam R. Chidamber and Chris F. Kemerer, "A Metric Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, Vol. 20, No. 6, June 1994.
- [ChuShe95]  
Neville I. Churcher and Martin J. Shepperd, "Comments on A metrics Suite for Object Oriented Design", *IEEE Transactions on Software Engineering*, Vol. 21, No. 3, March 1995.
- [ChuLee94]  
C.-M. Chung and M.-C. Lee, "Object-Oriented Programming Software Metrics", *International Journal on Mini and Microcomputers*, Vol. 16, No. 1, pp. 7-15, 1994.
- [Cop92]  
James O. Coplien, *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, ISBN 0201548550, 1992.

[Coo92]

William R. Cook, "Interfaces and Specifications for the Smalltalk-80 Collection Classes", *OOPSLA '92 Conference proceedings*, Vancouver, Canada, Oct. 18-22, ACM SIGPLAN Not. 27, 10, pp. 1-15, 1992.

[CouRo93]

Bernard Coulange and Alain Roan, "Object-Oriented Techniques at Work: Facts and Statistics", *TOOLS Europe'93 Conference proceedings*, pp. 89-94, 1993.

[Cri&a92]

John Cribbs, Suzanne Moon and Colleen Roe, "An Evaluation of Object-Oriented Analysis and Design Methodologies", *Alcatel Network Systems*, SIGS Books, 1992.

[DahNyg66]

Dahl, O. and Nygaard, K., "Simula, an Algol-based Simulation Language", *Communications of the ACM*, Vol. 9, pp. 671-678, 1966.

[Dav92]

Tsvi Bar-David, "Practical Consequences of Formal Definitions of Inheritance", *Journal of Object-Oriented Programming*, pp. 43-49, July/August 1992.

[DeM86]

Tom DeMarco, *Controlling Software Projects: Management, Measurement and Estimation*, Yourdon Press, ISBN 0-131717111, New York, September 1986.

[Dev96]

"Hybrid Hell", *Directions in desktop development*, Section Development, November 1996.

[Dic95]

H. Dicky, C. Dony, M. Huchard, T. Libourel, "ARES, Adding a Class and REStructuring Inheritance Hierarchy", *Onzièmes Journées Bases de Données Avancées*, Nancy, France, pp. 25-42, Aug/Sep 1995.

[DOD88]

DOD-STD-2167A/498, "The Waterfall model", *Defense Systems Software Development*, February 1988.

[Dum&a95]

Reiner R. Dumke, Erik Foltin and Achim S. Winkler, "Measurement-Based Quality Assurance in Object-Oriented Software Development", *OOIS '95 Conference proceedings*, pp. 315-319, December 1995.

[Ebe92]

Christof Ebert, "Correspondence Visualization Techniques for Analyzing and Evaluating Software Measures", *IEEE Transactions on Software Engineering*, Vol. 18, No. 11, pp. 1029-1-34, November 1992.

[Eli95]

Anton Eliëns, *Object-Oriented Software Development*, Addison-Wesley Publishers Ltd., ISBN 0-201-62444-3, 1995.

[Emb92]

Embley, D. W., B. D. Kurtz, and S. N. Woofield, *Object-Oriented System Analysis: A Model-driven Approach*, Prentice Hall, January 1992.

[Eng97]

Robert Englander, *Developing Java Beans (Java Series)*, O'Reilly & Associates Inc., ISBN 1565922891, June 1997.

- [Ewi94]  
Greg Ewing, *Class Inheritance: The mechanism and Its Uses*, Bachelor of Computing Thesis, Monash University October 1994.
- [Fen91]  
Norman E. Fenton, *Software Metrics - A Rigorous Approach*, Chapman and Hall, ISBN 0-412-40440-0, 1991.
- [Fir95]  
Donald Firesmith, "Inheritance guidelines", *Journal of Object-Oriented Programming*, pp. 67-72, May 1995.
- [Fol97]  
*The Free On-line Dictionary of Computing*, <http://wombat.doc.ic.ac.uk/>, Editor Denis Howe, <dbh@doc.ic.ac.uk>, 1997.
- [Foo89]  
Brian Foote and Ralph E. Johnson, "Reflective Facilities in Smalltalk-80", *OOPSLA '89 Conference proceedings*, pp. 327-335, October 1-6, 1989.
- [FooOpd94]  
Brian Foote and William F. Opdyke, "Life Cycle and Refactoring Patterns that Support Evolution and Reuse", *PLoP '94 Conference proceedings*, Monticello, Illinois, August 1994.
- [Fug&al98]  
Alfonso Fuggetta, Luigi Lavazza, Sandro Morasca, Stefano Cinti, Giandomenico Oldano and Elena Orazi, "Applying GQM in an Industrial Software Factory", *ACM Transactions on Software Engineering and Methodology*, October 1998.
- [Col&al94]  
Derek Coleman, Patrick Arnold, Stephanie Bodoff, Chris Dollin, Helena Gilchrist, Fiona Hayes, Paul Jeremaes, *Object-Oriented Development - The Fusion Method*, Prentice Hall International, 1994.
- [GabKem95]  
Gabriel Eckert and Magnus Kempe, "Modeling with Objects and Values: Issues and Perspectives", *ROAD (Report on Object Analysis & Design)*, Vol. 1, No. 5, pp. 20-27, 1995.
- [Gan77]  
C. Gane and T. Sarson, *Structured System Analysis: Tools and Techniques*, IST Inc., 1977.
- [Gam&al93]  
Erich Gamma, Richard Helm, Ralph Johnson John Vlissides, "Design Patterns: Abstraction and Reuse of Object-Oriented Design", *ECOOP '93 Conference proceedings*, Springer-Verlag Lecture Notes in Computer Science, 1993.
- [Gam&al95]  
Erich Gamma, Richard Helm, Ralph Johnson John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company, ISBN 0-201-63361-2, 1995.
- [Gib90]  
S. Gibbs, D. Tsihrizis, E. Casais, O. Nierstrasz and X. Pintado, "Class Management for Software Communities", *Communications of the ACM*, September. 1990, pp. 90-103.



[GolRob90]

Adele Goldberg and David Robson, *Smalltalk-80, the Language and its Implementation*, Addison-Wesley series in Computer science, ISBN 0-201-11371-6, 1985.

[Gra92]

Justin O. Graver, "T-gen: a string-to-object translator generator", *Journal of Object-Oriented Programming*, pp. 35-42, University of Florida, Computer and Information Sciences, e-mail: graver@ufl.edu, September 1992.

[Gra94]

Ian Graham, *Object Oriented Methods*, Second edition, Addison-Wesley, ISBN 0-201-59371-8, 1994.

[HadGeo95]

H.M. (Al) Haddad and K.M. George, "A Survey of Method Binding and Implementation Selection in Object-Oriented Programming Languages", *Journal of Object-Oriented Programming*, October 1995.

[HarNit96]

Rachel Harrison and R. Nithi, "An Empirical Evaluation Of Object-Oriented Design Metrics", *OOPSLA '96 Conference proceedings*, Workshop on "OO Product Metrics", Workshop report, 1996.

[Hen95]

Brian Henderson-Sellers , "OO Metrics Programme", *Object Magazine*, pp. 73-79, 95, October 1995.

[Hen96]

Brian Henderson-Sellers, *Object-Oriented Metrics, Measures of Complexity*, Prentice Hall Object-Oriented Series, ISBN 0-13-239872-9, 1996.

[HenEdw94]

Brian Henderson-Sellers and Julian Edwards, *BookTwo of Object-Oriented Knowledge - The Working Object*, Prentice Hall, ISBN 0-13-093980-3, 1994.

[HenNyq92]

FN/Mats Henricson and Erik Nyquist, "Programming in C++, Rules and Recommendations", *Ellemtel Telecommunication Systems Laboratories*, Document No. M 90 0118 Uen, 1992.

[Hit95]

Martin Hitz, "Measuring Reuse Attributes in Object-Oriented Systems", *OOIS '95 Conference proceedings*, pp. 19-38, December 1995.

[HitFir97]

Michael Hitchens and Andrew Firmage, "The Design of a Flexible Class Library Management System", *TOOLS24 '97 Conference Proceedings*, Beijing, China, pp. 71-80, November 1997.

[HitMon95a]

Martin Hitz and Behzad Montazeri, "Measuring Coupling and Cohesion In Object-Oriented Systems", *International Symposium on Applied Corporate Computing*, Monterrey, Mexico, October 1995.

[HitMon95b]

Martin Hitz and Behzad Montazeri, "Measuring Product Attributes of Object-Oriented Systems", In Schäfer and P. Botella (eds.), *ESEC '95 proceedings, 5<sup>th</sup> European Software Engineering Conference*, Barcelona, Spain, pp. 24-136, LNCS989, Springer-Verlag, September 1995..

[HitMon96]

Martin Hitz and Behzad Montazeri, "Chidamber & Kemerer's Metrics Suite: A Measurement Theory Perspective", *IEEE Transactions on Software Engineering*, pp. 267-271, Vol 22, No 4, April 1996.

[Hop94]

Trevor P. Hopkins, "Complexity Metrics for Quality Assessment of Object-Oriented Design", *Software Quality Management '94 Conference proceedings*, pp. 467-481, 1994.

[Ibe94]

IBEX Computing SA, "ITASCA Object Database Management System, DBA Manual", International Business Park, 4ème Blvd, Bât Héra, F-74160 Archamps, France, <http://www.ibex.ch>, 1994.

[Jac83]

M.A., Jackson, *Jackson System Development*, Englewood Cliffs, Prentice-Hall Inc., 1983.

[Ke92]

Brian M. Kennedy, "Design for Object-Oriented Reuse in the OATH Library", *Journal of Object-Oriented Programming*, pp. 51-57, Jul/Aug 1992.

[Kem96]

Chris F. Kemerer, "Measuring the Unmeasurable", *Bournemouth Metrics Workshop 1996*, Bournemouth University.

[KosVih92]

Kai Koskimies and Juha Vihavainen, "The problem of Unexpected Subclasses", *Journal of Object-Oriented Programming*, pp. 53-59, October 1992.

[Kow93]

Ralph Kolewe, "Metrics in Object-Oriented Design and Programming", *Software Development Journal*, Vol. 1, pp. 53-62, October 1993.

[LalPug91]

Wilf Lalonde and John Pugh, "Subclassing  $\neq$  Subtyping  $\neq$  Is\_a", *Journal of Object-Oriented Programming*, pp. 57-62, January 1991.

[LalPug94]

Wilf Lalonde and John Pugh, "Gathering Metric Information Using Metalevel Facilities", *Journal of Object-Oriented Programming*, pp. 33-37, Mar/Apr 1994.

[Lee&al93]

Yen-Sung Lee, Bin-Shiang Liang and Feng-Jian Wang, "Some Complexity Metrics for Object-Oriented Programs Based on Information Flow", *IEEE 0-8186-4030-8*, pp. 302-310, 1993.

[Lew95a]

John A. Lewis, "Quantified Object-Oriented Development: Conflict and Resolution", *4th Software Quality Conference*, pp. 220-229, Vol. 1, University of Abertay, Dundee, July 4-5, 1995.

[Lew95b]

Simon Lewis, *The Art and Science of Smalltalk*, Prentice Hall/Hewlett-Packard Professional Books, ISBN 0-13-371345-8, 1995.

[Lis87]

Barbara Liskov, "Data Abstraction and Hierarchy", *OOPSLA '87 Addendum to the proceedings*, May 88, pp. 17-34, October 1987.

- [Liu96]  
Chamond Liu, *Smalltalk, Objects and Design*, Manning Publications Co., ISBN 1-884777-27-9, 1996.
- [LiHen93]  
Wei Li and Sallie Henry, "Object-Oriented Metrics Which Predict Maintainability", *Journal of Systems and Software*, Vol. 23, No. 2, pp. 117-122, 1993.
- [LinRüp95]  
Thomas Lindner and Andreas Rüping, "How Formal Object-Oriented Design Supports Reuse", *Architectures and Processes for Systematic Software Construction*, Eduardo Casais (Ed.), FZI-Publication 1/95, Forschungszentrum Informatik Karlsruhe, 1995
- [LisGut86]  
Barbara Liskov and John Guttag, "Abstraction and Specification in Program Development", *MIT Press*, Cambridge (Mass.), 1986.
- [LorKid94]  
Mark Lorenz and Jeff Kidd, *Object-Oriented Software Metrics*, Prentice Hall Object Oriented Series, Englewood Cliffs (N.J.), 1994.
- [MadMøl93]  
Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard, *Object-Oriented Programming in the Beta Programming Language*, Addison-Wesley, ISBN 0-201-62430-3, 1993.
- [Mar78]  
T. De Marco, *Structured Analysis and System Specification*, Yourdon, Inc., 1978.
- [McG95]  
John D. McGregor, "Managing Metrics in an Iterative Environment", *Object Magazine*, pp. 65-71, October 1995.
- [McKMon93]  
James C. McKim, Jr. and David A. Mondou, "Class Interface Design: Designing for Correctness", *Journal of Systems and Software*, No. 23, pp. 85-94, 1993.
- [Mel95]  
Stephen J. Mellor, "Reuse Through Automation: Model-Based Development", *Object Magazine*, pp.50-55, September 1995.
- [Mey88]  
Bertrand Meyer, *Object-oriented Software Construction*, Prentice Hall International, C.A.R. Hoare, Series Editor, ISBN 0-13-629049-3, 1988. <http://www.eiffel.com>.
- [Mey97]  
Bertrand Meyer, *Object-oriented Software Construction - Second Edition*, Prentice Hall PTR, ISBN 0-13-629155-4, 1997. <http://www.eiffel.com>.
- [Mey&a195]  
Bertrand Meyer, Éric Bezault, David Morgan and Xavier Le Vouch, "ISE Eiffel: The Environment", ISE Technical Report TR-EI39/IE, Interactive Software Engineering Inc. (ISE), 1995. <http://www.eiffel.com>.
- [MonPuh92]  
David E. Monarchi and Gretchen I. Puh, "A Research Typology for Object-Oriented Analysis and Design", *Communications of the ACM*, Vol. 35, No. 9, pp. 35-47, September 1992.

[MorDom89]

Dennis R. Moreau and Wayne D. Dominick, "Object-Oriented Graphical Information Systems: Research Plan and Evaluation Metrics", *The Journal of Systems and Software*, Vol. 10, pp. 23-28, 1989.

[New&a196]

Alexander Newman and al., *Special Edition, Using Java*, Que Corporation, ISBN 0-7897-0604-0, 1996.

[ObjSha93]

Objectshare Systems Inc. and Dan Shafer, *WindowBuilder Pro/V 1.0 - Tutorial and Reference Guide*, *Objectshare Systems Inc.*, October 1993.

[Ode92]

James J. Odell, "Managing Object Complexity, part I: Abstraction and Generalization", "Managing Object Complexity, part I: Composition", *Journal of Object-Oriented Programming*, September and October 1992.

[Ode94]

James J. Odell, "Six different kinds of Composition", *Journal of Object-Oriented Programming*, pp. 10-15, January 1994.

[OOP93]

*OOPSLA '93, Workshop #19 - "Understanding Object-Model Concepts"*, 1993.

[Owe95]

John Owens, *Using Object-Oriented Databases To Model Hydrocarbon Reservoirs*, PhD Thesis, University of Aberdeen, 1995.

[Pap95]

David M. Papurt, *Inside the Object Model - The Sensible Use of C++*, SIGS Books, ISBN 1-884842-05-4, 1995.

[PapLeJ97]

David M. Papurt and Jean Pierre LeJack, "The Sensible Use of Method Overriding", *Journal of Object-Oriented Programming*, pp. 62-65, 71, Mar/Apr 1997.

[Ped89]

Claus H. Pedersen, "Extending Ordinary Inheritance Schemes to Include Generalization", *OOPSLA '89 Conference proceedings*, pp. 407-417, October 1-6, 1989.

[Pol97]

Geert Poels, "An Analytic Evaluation of Static Coupling Measures for Domain Object Classes", *ECOOP '98, Object-Oriented Product Metrics for Software Quality Assessment Workshop*, Brussels, Belgium, July 21, 1998.

[Pow98]

Powersoft, Sybase Inc., *PowerBuilder User's Guide*, Open Tools from Sybase Inc., <http://www.powersoft.com/products/powerbuilder/pb6proinfo.html>, 1998.

[Rad93]

Klaus Radermacher, "Abstraction Techniques in Semantic Modelling", *Information Modelling and Knowledge Bases IV*, H. Jaakkola et al. (eds.), IOS Press Amsterdam 1993.

[Rie96]

Arthur J. Riel, *Object-Oriented Design Heuristics*, Addison-Wesley Publishing Company, ISBN 0-201-63385-X, 1996.

[Riv96]

Fred Rivard, "Smalltalk: a Reflective Language", *Reflection '96 proceedings*, pp. 21-38, edited by Gregor Kiczales, San Francisco, USA, Avril 1996.

[Rob92]

Teri Roberts, "Metrics for Object-Oriented Software Development", Workshop report, *Addendum to the OOPSLA '92 proceedings*, pp. 97-100, October 1992.

[RobBol89]

Pierre N. Robillard and Germinal Boloix, "The Interconnectivity Metrics: A New Metric Showing How a Program Is Organised", *The Journal of Systems and Software*, Vol. 10, pp. 29-39, 1989.

[RosHya96]

L. Rosenberg and L. Hyatt, "Developing a Successful Metrics Program", Proceedings of 2nd Annual Conference on Software Metrics, Washington, DC, June 1996.

[Rum91]

James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenzen, *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs (N.J.), 1991.

[Rum93]

James Rumbaugh, "On the Horns of the Modeling Dilemma - Choosing Among Alternate Modeling Constructs", *Journal of Object-Oriented Programming*, pp. 8-17, Nov/Dec 1993.

[Rum96]

James Rumbaugh, "A Matter of Intent: How to Define Subclasses", *Journal of Object-Oriented Programming*, pp. 5-9, 18, September 1996.

[Sca&al93]

Bruno Scardia, Antoine Proietto and Thierry Labbé, "ORGUE: un Outil pour la Conception des Logiciels Temps Réel selon la Méthode TOCCATA", *Génie Logiciel & Systèmes Experts*, No. 32, pp. 24-30, September 1993.

[Sch98]

Schäffer B., "Design and Implementation of Smalltalk Mixin Classes", *Ubilab Technical Report*, 98.11.1, UBS AG, Zürich, Switzerland, November 1998.

[SEL95]

Software Engineering Laboratory, "Software Measurement Guidebook", NASA, Goddard Space Flight Center, Greenbelt (MD) SEL-94-102, June 1995.

[Sei96]

Ed Seidewitz, "Controlling Inheritance", *Journal of Object-Oriented Programming*, pp. 36-42, January 1996.

[Sha92]

Dan Shafer, Scott Herndon and Laurence Rozier, *Smalltalk Programming for Windows*, Prima Publishing, Rocklin, CA, 1992.

[Sha96]

Bruno Shafer, "Advanced Smalltalk: Elegance and Efficiency", *10th European Conference on Object-Oriented Programming*, Tutorials Notes, July 8-12, Linz, Austria, 1996.

[She90]

Martin Shepperd, "Early life-cycle metrics and software quality models", *Information and Software Technology*, pp. 311-316, 1990.

[She&al91]

Sheetz, Steven D., David P. Tegarden, and David E. Monarchi, "Measuring Object-Oriented System Complexity", Workshop on Information Technology and Systems - WITS-91, 12/91.

[Sho&al93]

Jean Scholtz, Shyam Chidamber, Robert Glass, Al Goerner, Mary Beth Rosson, Mike Stark and Iris Vessey, "Object-Oriented Programming: The Promise and the Reality", *Journal of Systems and Software*, Vol. 23, pp. 199-204, 1993.

[SimLew98]

Frank Simon and Claus Lewerentz, "A Product Metrics Tool Integrated Into a Software Development Environment", *12th ECOOP '98, Object-Oriented Product Metrics for Software Quality Assessment Workshop*, CRIM Montreal, ISBN 2--921316-87-0, pp. 36-41, 1998.

[Smi91]

David N. Smith, *Concepts of Object-Oriented Programming*, McGraw-Hill series on programming languages, ISBN 0-07-059177-6, 1991.

[Ste90]

Guy L. Steele, *Common Lisp the Language, 2<sup>nd</sup> edition*, Thinking machines Inc., Digital Press, ISBN 1-55558-041-6, 1990.

[Ste&al96]

Patrick Steyaert, Carine Lucas, Kim Mens and Theo D'Hondt, "Reuse Contracts: Managing the Evolution of Reusable Assets", *OOPSLA '96 Conference proceedings*, 1996.

[Str90]

Stroustrup B. Ellis M., *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[Sun99]

Sun Microsystems Inc., *Java™ 2 SDK, Standard Edition Documentation, Version 1.3*, 1999.

[Tai96]

Antero Taivalsaari, "On the Notion of Inheritance", *ACM Computing Surveys*, Vol. 28, No. 3, pp. 439-479, September 1996.

[Tai98]

Antero Taivalsaari, "Implementing a Java Virtual Machine in the Java Programming Language", Sunlabs Microsystems, Technical Report, SMLI TR-98-64, March 1998.

[Tea&al96]

Java Team, James Gosling, Bill Joy and Guy Steele, *The Java™ Language Specification*, part of The Java™ Series, Sun Microsystems, Inc., ISBN 0-201-63451-1, April 1996.

[Teg&al95]

David P. Tegarden, Steven D. Sheetz and David E. Monarchi, "A Software Complexity Model of Object-Oriented Systems", *Decision Support Systems: The International Journal*, No. 13, pp. 241-262, 1995.

[VerCor95]

Frans Ververs and Cornelis Pronk, "On the Interaction between Metrics and Patterns", *OOIS '95 Conference proceedings*, pp. 303-314, December 1995.

[OMG97]

Object Management Group, "Unified Modeling Language, version 1.1, Updated September 1, 1997", *Rational Software*, <http://www.rational.com>, 1997.

[Wan88]

Yair Wand, "An Ontological Foundation for Information Systems Design Theory" and "A Proposal for a Formal Model of Objects", *Proceedings of the IFIP WG 8.4. Working Conference on Office Information System: The Design Process*, Linz, Austria, August 1988.

[Web95]

Bruce F. Webster, *Pitfalls of object-oriented development*, ISBN 1-55851-397-3, 1995.

[Whi96a]

Scott A. Whitmire, "A Formal Object Model for Measurement", *OOPSLA '96 Conference proceedings*, Workshop on "OO Product Metrics", Workshop report, 1996.

[Whi96]

Robin Whitty, "Object-Oriented Metrics: An Annotated Bibliography", *SIGPLAN Notices*, <http://www.sbu.ac.uk/~csse/publications/OOMetrics.html>

[Whi97]

Scott A. Whitmire, "Object-Oriented Design Measurement", John Wiley & Sons, Inc, ISBN 0-471-13417-1, 1997.

[Wil93]

John D. Williams, "Metrics for Object-Oriented Projects", *Object Expo Europe Conference Proceedings*, SIGS Publications New York, USA, July 1993.

[Wil96]

Willis C. P., "Analysis of Inheritance and Multiple Inheritance", *Software Engineering Journal*, Vol. 11, No. 4, pp. 215-224, July 1996.

[X3J96]

X3J20, Smalltalk committee, 1996.

[You79]

E. Yourdon and L. Constantine, *Structured Design*, Englewood Cliffs, Prentice-Hall Inc., 1979.

[ZhaOve97]

T.C. Zhao and Mark Overmars, *Forms Library - A Graphical User Interface Toolkit for X V0.88*, <http://www.westworld.com/~dau/xforms/forms.index.shtml>, November 1992.

## A. Appendix

### A.1. Heuristics' classification

Categories of heuristics [Fir95, HenEdw94, Mey88, Pap95, Rie96] are organised according to the main aspects of OO modelling as follows:

Categories	Topics
OO conceptual model	The object model Abstraction, abstract data types
Architecture from objects to systems	Encapsulation and information hiding Class-specific data and behaviour Responsibilities, roles, contracts, interfaces
Modularity and subsystems	Classes relationships and objects coupling Communication, message-passing Relationships: association, aggregation
The inheritance relationship and class hierarchy	Generalisation/specialisation Inheritance identification Reuse Multiple inheritance Physical OO design

### Heuristics for subclass's definition [Rum96]

Topics	Heuristics
Full inheritance	A subclass should inherit all properties from its superclass without restricting or deleting
Extension	A subclass should add further features to the ones inherited
Behaviour compatibility	A subclass should either: <ul style="list-style-type: none"> <li>• Reuse without change</li> <li>• Implement the declared deferred method</li> <li>• Be a combination of inherited behaviour and new functionality e.g. self in Smalltalk or before-and-after in CLOS</li> <li>• Override with extreme care</li> </ul>
Form change	A subclass must have a different structure from its superclass e.g. additional attributes, associations
Restriction	A subclass should not restrict the inherited properties

### Summary of Fusion's method guidelines [Fus94]

Topics	Heuristics
Class definition	<ul style="list-style-type: none"> <li>• Properties must describe all instances of the class</li> <li>• A class must represent one and only one abstraction</li> <li>• A class should be cohesive</li> <li>• An operation should perform a single function</li> </ul>
Object interactions	<ul style="list-style-type: none"> <li>• Reduce the coupling between objects</li> <li>• Reduce objects' dependencies</li> <li>• Objects should be organised into independent sub-systems</li> </ul>
Use of inheritance	<ul style="list-style-type: none"> <li>• Abstract common properties in abstract classes</li> <li>• Avoid implementation inheritance</li> <li>• Polymorphism is recommended when the semantics of the inherited operations remain the same</li> <li>• Develop the class hierarchy in depth instead of width</li> <li>• Root should be defined as an abstract class</li> <li>• Each sibling should be semantically different</li> <li>• Preserve subtype inheritance</li> <li>• Behavioural subtyping should be preserved even if inheritance is a code-reuse mechanism and not a subtyping facility</li> </ul>



## A.2. Detailed design of the main components of the metric prototype tool

### A.2.1. Basic metrics repository

During the process of determining the lineage of a method, it is implicitly assumed that all parent-children relationships are known, however this is not straightforward. In particular, the amount of specific development for pattern matching varies depending on how much information can be directly obtained from the environment and the language used. In most cases, it would require a minimum amount of code analysis. In [SimLew98], the authors proposed a generic model that deals with all the language specifics however it also necessitates the use of a scripting language for describing the metrics to implement. Smalltalk provides a native set of functions permitting easy querying of the system for meta-information. Table A.1 shows examples of such features.

Category	Smalltalk command	Returned values
Organisational	aClass superclass	The direct parent class of aClass
	aClass subclasses	The direct subclasses of aClass
	aClass allSuperclasses	All parent classes of aClass
	aClass allSubclasses	All subclasses of aClass
Class description	AnObject class	The class name of anObject
	AClass allInstances	All instances of aClass
	AClass selectors	All methods of aClass
	AClass allInstVarNames	All instance variables of aClass
	AClass allClassVarNames	All class variables of aClass
Coupling	ASymbol implementors	All methods that provide an implementation of aSymbol
	ASymbol senders	All methods that sends a aSymbol message

Table A.1: Smalltalk metaclass information

Some of the features presented in the Table A.1 are candidate metrics themselves. When a Smalltalk command returns a set of objects, the use of the command `aSet size` returns the number of objects in the set. A metric is referred to as *basic* in the sense that it represents a simple counting of a feature of the implementation. Although such metrics may be useful as indicators of size, they are often utilised to form more complex metrics to address a particular aspect of the design e.g. the redefinition metrics. Metrics repositories can be used as a catalogue of measures for various purposes. Although the metric collector tool does not deal with the management of metric's definition, a possible repository structure may include the name, the definition of the metric e.g. Smalltalk commands, the description, its uses, its meaning and other related properties. In the experiments, the associated values of basic metrics were mostly of interest as they were often utilised during the derivation process. Rather than re-computing a metric every time it is needed, the metrics results are stored in the repository. The time processing aspect should not be neglected during the metrics collection. Because of the nature of the processing involved for the derivation of metrics e.g. inheritance structure parsing and computation, a pre-calculation is adopted whenever possible.

As Smalltalk provides an exporting functionality that enables persistent storage of objects on disk, there is no need to transform the metric repository structures and their values, as they are

themselves objects. Such direct mapping between memory and disk is convenient, as it does not add much additional development cost for the metric tool.

### A.2.2. Dictionary structures for metrics

The derivation process can be separated in two phases:

1. Collection of design information from all the classes included in the derivation.
2. Calculation of the redefinition metrics based on data previously collected.

While the second phase consists of the application of the metrics formulas, the first phase is an essential preparation phase where information is gathered and organised for later use. The description of this first phase follows.

An important aspect of the design is the use of simple repository structures that hold intermediary or final results for metrics. Before the computation of the metrics, a preparation phase gathers and temporarily stores all necessary information into Dictionary<sup>30</sup> objects in memory. This structure permits a rapid access to the metrics values. As the values of a Dictionary object can themselves hold references to other Collection objects, it is therefore possible to build flexible multi-dimensional dictionaries or Collection objects.

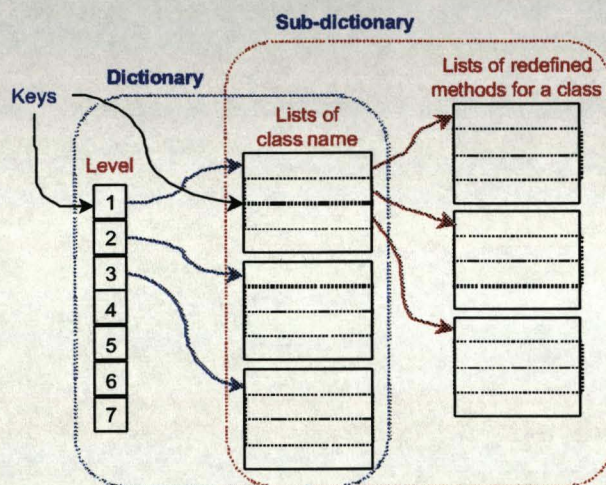


Figure A.1: Dictionary of redefined methods per class

As the redefinition metrics are calculated in relation to the hierarchy level, the metric tool gathers a list of classes to be included in the derivation at the corresponding level. In Figure A.1, a Dictionary object is used to store the class and method names. The levels in the hierarchy are the dictionary keys and the corresponding values are the class names. In order to store the list of redefined methods for each class, it is appropriate to use a dictionary type object with the class

<sup>30</sup> Similar to an indexed table structure, Dictionary objects in Smalltalk are Collection objects. They hold a key that enables direct access to an associated value.

names being values for the main dictionary as well as keys for the sub-dictionary. Therefore, the corresponding values for the sub-dictionary are the method names. In the experiments with the Smalltalk class hierarchy, the maximum depth level is seven. Although it is necessary to search for inheritance relationships between the classes for determining if a method is redefined, this information is not recorded in the dictionary structure.

Given the information in Figure A.1, the derivation of the metrics still necessitates the calculation of the following:

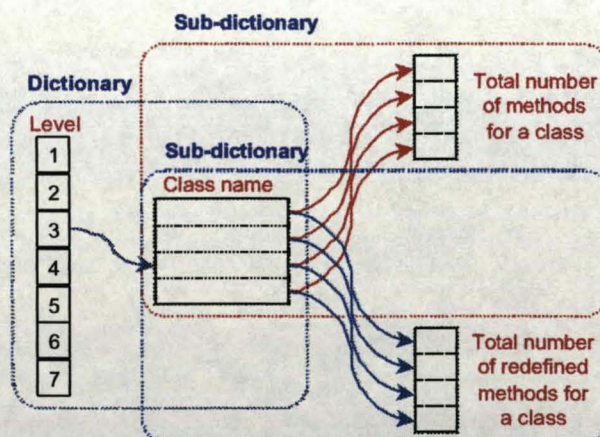


Figure A.2: Dictionary for the total number of methods per class

In Figure A.2, two different dictionaries with identical entry structure are represented. One sub-dictionary is used for the total number of methods for a class while the other is for the total number of redefined methods. The values are calculated directly from the dictionary in Figure A.1 or during the parsing of the classes. Although such numbers can be calculated at request time, their pre-calculations are often useful for a quick review of the metrics results. In such a way, if no updates have been done on the classes assessed, no re-calculations are needed and expensive parsing is avoided.

Note that, during the parsing of the classes, in the case of the calculation of the PRMC (see 3.2), the total number of methods for a class is being cumulated with those inherited to form the new total number of methods. The user sets the cumulative option in the collector tool before the request for metrics derivation.

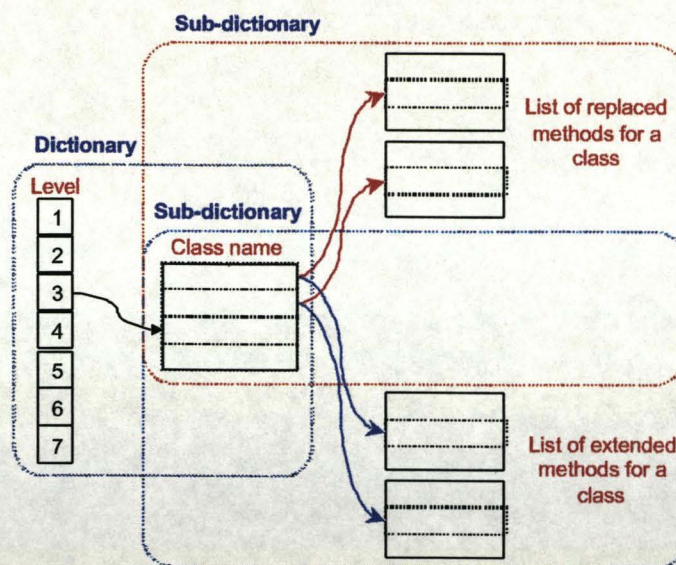


Figure A.3: Dictionary for replaced and extended methods

So far, the information gathered in the above dictionaries permits the calculation of the general PRM regardless of the method's state. In Figure A.3, two more dictionaries, again with identical entry structure, shows for each class, the list of methods being replaced or extended. In the same manner than as in Figure A.2, dictionaries for the total number of replaced and extended methods can be built. Thus, the preparation phase is complete and the second phase of the derivation process can take place.

### A.2.3. A persistent repository structure

Persistently storing metrics values is an important feature for enhancing the metric tool capabilities. The persistent features, not only improves the usability of the tool but also opens a wide-range of possibilities for further development such as the management of metrics results versions for comparison of design versions. Figure 4.1 shows only the relevant, adapted classes and methods taken from [Owe95].

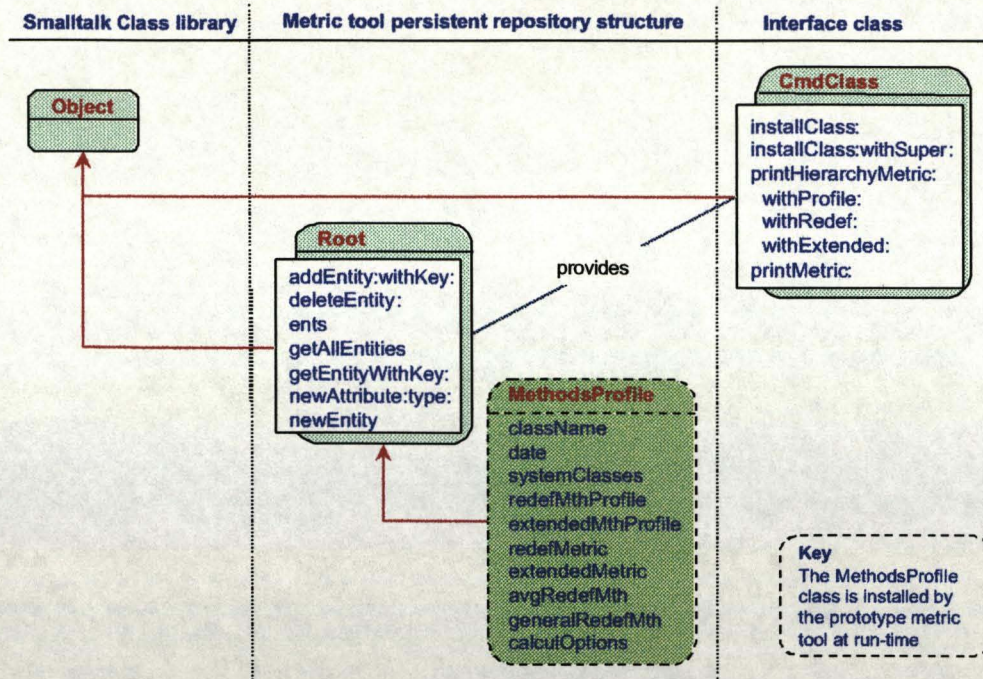


Figure A.4: Persistent repository model

The main purpose of the PR is to provide features for the management of persistent objects. The main particularity of the system is that all the persistent objects are dealt and stored within the Smalltalk image in exactly the same manner as any other 'live' objects at any one time in the environment. This provides a uniform and coherent access to both control or data objects from the metric tool prototype. The two main classes `Root` and `CmdClass` inherit from the `Object` class and provide the necessary functionalities for method profile management. The PR system permits dynamical installation of classes within the image. The `CmdClass` class methods are the interfaces to the PR e.g. the self explanatory `installClass:withSuper:` method and uses the low-level `Root` class methods. For objects to be persistent, the corresponding class has to be first created and installed within the Smalltalk environment by the PR. Such a class is subclass of the `Root` class. In such a way, the `Root` class generic methods are inherited by any of its subclasses. Note that generic methods are methods that manage the persistent objects. In the PR terminology, (see `Root` class's methods in Figure A.4), a method profile object is referred to as *an entity* that holds a set of attributes. In addition, dynamic changes to an inserted class are possible e.g. deletion or addition of new attributes to a class. A key string that acts as an object identifier allows uniqueness and access to objects. The main changes from the PR original version includes the adaptation of some methods to take into account the metric requirements. Most of the changes are low-level changes such as the location of temporary stored files. These did not affect the original interface functions in the `CmdClass` class. Extension of functionalities was realised by addition of new methods in this class e.g. printing facilities.

The major benefits of the use of the PR system is that classes and objects can be dynamically created and recalled regardless of the underlying storage mechanism. The PR also provides various

class methods for the management of objects. Treated as objects, the method profiles include the list of attributes in Figure A.4 i.e. mainly the dictionaries described in section A.2.1. The following section describes the installation of the `MethodsProfile` class by the profile manager component.

#### A.2.4. The profile manager

The profile manager (`MethodsProfileList` class) is the core component that supervises the derivation process (Figure A.5). It receives the requests from the user interface and verifies if the requests have not been previously processed i.e. existence of method profile objects. If it is the case, only a re-calculation of the metrics is necessary i.e. second phase of the derivation process, therefore the metric results displayed correspond to a previous measure. For an update of the measures, the user should issue an explicit request within the profile interface browser. Before the launch of the derivation process, the profile manager should ensure that a `MethodsProfile` class exists to proceed further. To do so, the `initMethodsProfile` method in `MethodsProfileList` class places requests to the PR via the `ProfileDBAPI` interface methods. It should be noted that the profile manager has been specifically developed to provide support for the assessment of class hierarchies. Therefore, the concerned method profiles are mainly classes organised as a tree hierarchy i.e. branches or entire class hierarchy (see experiments with the redefinition metric at system level in section 0).

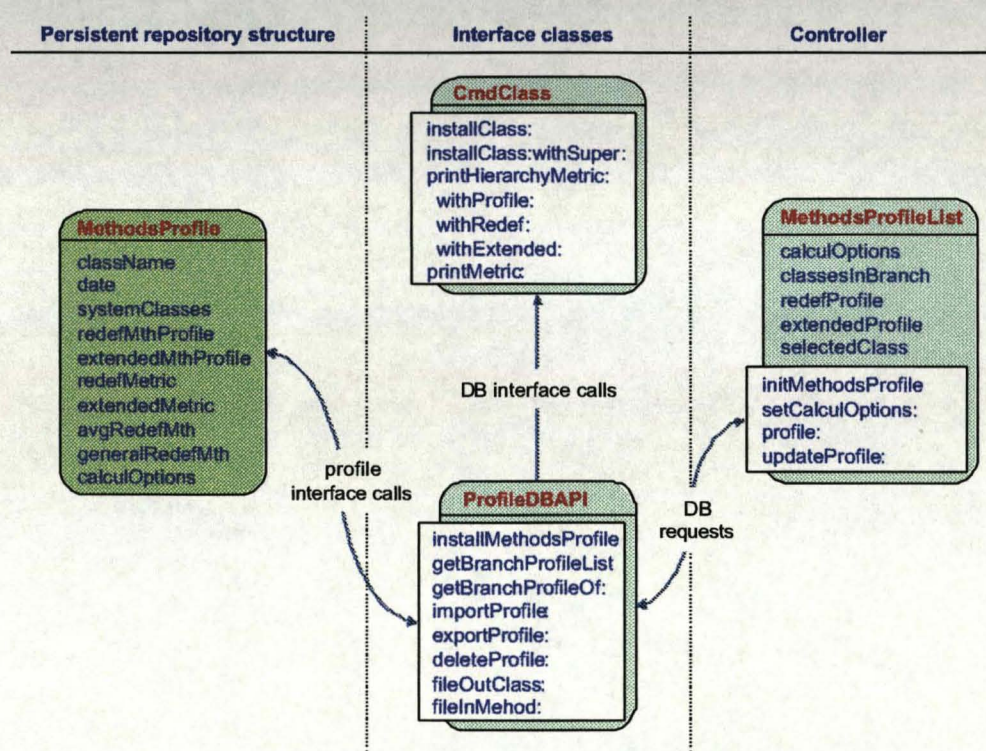


Figure A.5: Profile manager model

The installation of the `MethodsProfile` class is a one-off task. Given a class description i.e. class name and an attributes list, the `installMethodsProfile` method in the `ProfileDBAPI` class

automatically generates a subclass of the `Root` class called `MethodsProfile`. For each of the attributes created, a corresponding accessor method is automatically generated in the new installed class, thereby enabling future consultation of the objects attribute values. The list of attributes includes:

- Several dictionaries (see section A.2.1) detailing the life history of extended and redefined methods (respectively the `extendedMthProfile` and `redefMthProfile` attributes).
- The corresponding `PCRM` and `PEM` values (respectively the `extendedMetric` and `redefMetric` attributes).
- Other relevant information that defines the context of derivation such as the `date` and the `calculOptions` attributes.

At run-time, the profile manager (`MethodsProfileList` class) maintains a list of existing profiles i.e. `MethodsProfile` objects in memory. Any update of a method profile is preceded by a deletion of the `MethodsProfile` object before the start of the entire derivation process. For this reason, it is important to date-stamp the derivation process at the original date of request. Notice that a finer-grained stamping method may be possible e.g. time.

The metric derivation and the method profiles building activities share common parsing tasks. Despite a potential additional processing time, both activities are realised within a same functionality. In all cases, the availability of the method profiles is essential during the analysis and interpretation phase.

Whether the derivation of the metrics is requested for a class, a branch of the hierarchy or a system, a unique identifier is used for naming and storing the method profiles built. By default, the top node class for a branch of the hierarchy is the identifier in the case of metrics applied at hierarchy level. For a class and a system, respectively, the class name and an arbitrary name acts as identifier.

The `calculOptions` attribute is initialised by the `setCalculOptions:` method, both in the `MethodsProfileList` class. This attribute holds the desired derivation options as well as the control options for internal purposes e.g. display options. For the metric collector tool, only two options are relevant:

- The cumulative option: used for the calculation of the cumulative redefinition metric.
- The compiler classes inclusion: in Smalltalk, the compiler classes are hidden classes [GolRob85] and are only accessible on explicit request. This option offers the possibility to do so. As these classes are special internal classes, by default they were not included during the experiments.

The `calculOptions` attribute can be a placeholder for further options. Being a dictionary type object, the use of this attribute is flexible and can be extended for further requirements. The option values are saved in the method profile object as well.

### A.2.5. The metric engine

The metric engine (`RedefMetric` class) incorporates the necessary parsing and calculation algorithms for the redefinition metrics (Figure A.7). It is the profile manager object that initiates the creation of a metric engine object. Once a method profile object has been initialised, the derivation request is passed on to the metric engine object by the profile manager object for processing. A metric engine object stores temporarily information in its attributes, gathered during the course of parsing. Only on completion of the processing tasks, does the method profile object regain control and transfer the results to the corresponding method profile object. In such a way, the derivation request is completely delegated to the metric engine object and its lifetime lasts while the method profile is built.

### A.2.6. The hierarchy browser and profile manager designs

The main user interface integrates a similar hierarchy browser as the one provided by the Smalltalk environment and a tabular set of fields for the display of the metric results. The maximum display of levels of the hierarchy is fixed to seven for convenience reasons. When the prototype metric tool is running, an instance of the `SystemMetricBrowser` class is created and represents the main interface window (see Figure A.6). In the case of a class or hierarchy metric request, a dialog box is presented to the user for entering the name of the class concerned. Then, the metric browser object directly creates an instance of the profile manager object and continues the derivation process.

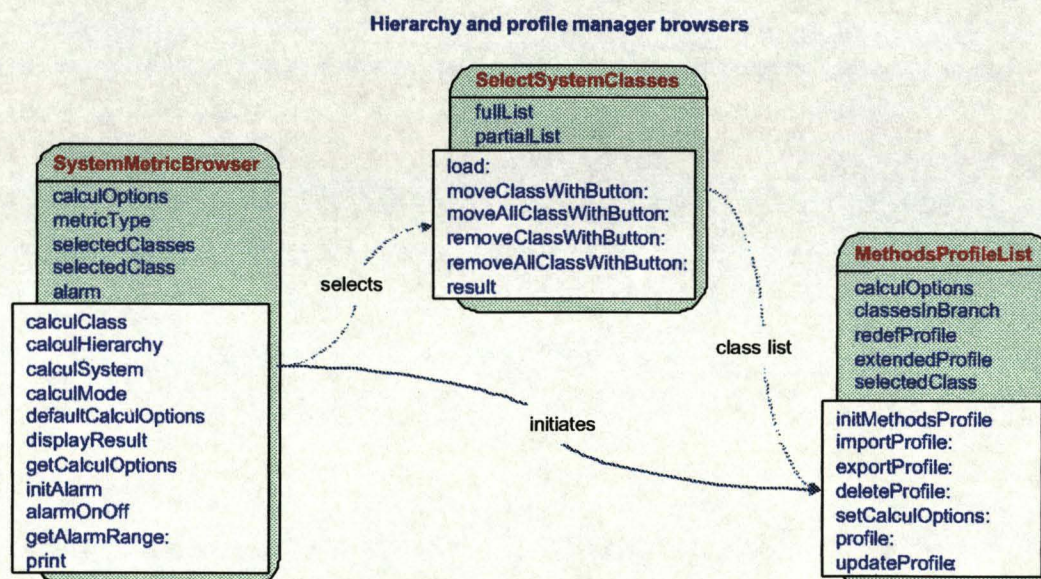


Figure A.6: The hierarchy browser and profile manager designs



The `MethodsProfileList` class plays both the role of profile manager and the interface for profile management. Although an instance of this class is always created for metric processing, it only interacts with the user on request of the profile management function. The user interface permits the deletion, update, view, import, export and print of an existing list of profiles currently stored (see the corresponding methods in the `MethodsProfileList` class in Figure A.6).

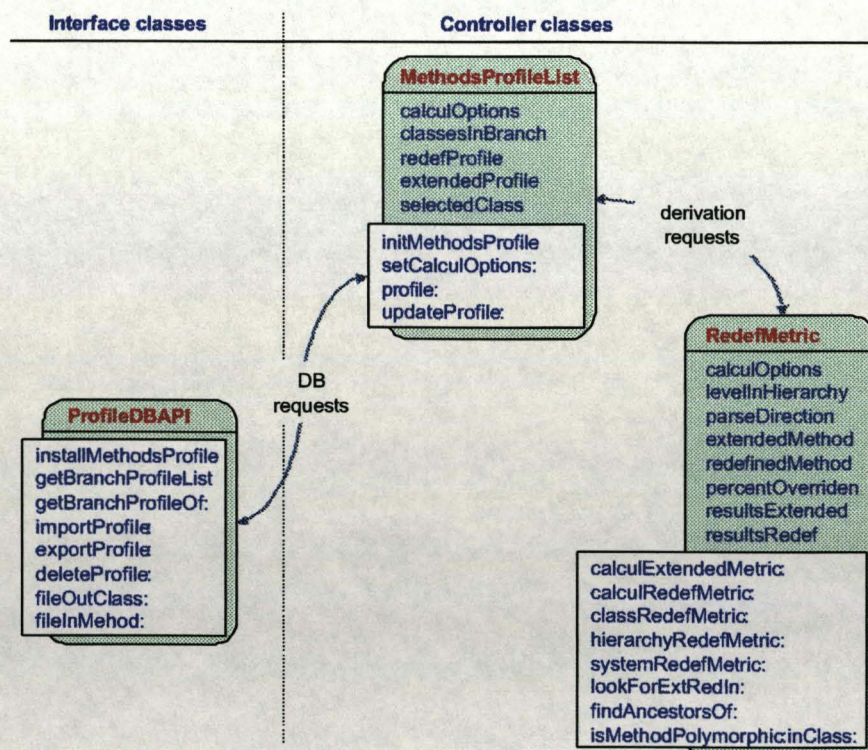


Figure A.7: Metric engine model

The main function of the metric engine object is to search for redefined methods within a given set of classes. During the parsing of the classes, the object constructs four main dictionaries (as its object attributes):

- The `extendedMethod` attribute which stores all extended methods per class per level.
- The `redefMethod` attribute which stores all replaced methods per class per level.
- The `resultsExtended` attribute which stores the percentages for extended methods per level.
- The `resultsRedef` attribute which stores the percentages for replaced methods per level.

To determine if a method is redefined in a class *X*, the `findAncestorsOf:` method looks up for the list of parent classes of the given class. The presence of the method signature in, at least one of *X*'s parent classes, permits conclusion that the present method is redefined. However, further analysis is required to detect in which case of redefinition the method falls under. The `hierarchyRedefMetric:` method is the main entry point to the parsing algorithm for the metric at hierarchy level. The `classRedefMetric:` and the `systemRedefMetric:` methods are respectively, the methods for calculating the metrics at class and system levels. The profile manager object

invokes them all, in addition to the `lookForExtRedIn:` method that determines whether a method is being extended or replaced. Updates of the methods profile are done accordingly. Additional useful information is the detection of methods originally declared as *polymorphic*. To do so, the `isMethodPolymorphic:inClass:` method examines the method source code for the Smalltalk `implementedBySubclass` pattern.

On completion of the calculations, the profile manager object requests the dictionary object identifiers built by the metric engine object and reassigns them to the corresponding method profile objects.

The algorithms can be decomposed in two phases:

- The search for redefined methods.
- The search for the type of redefinition used.

The main difference between the two phases lies in the information searched. If such a metric was to be applied early in the development process, it is assumed that, at design phase, the method signatures would be known, therefore this information would be sufficient to realise the first phase of the algorithm. The body of the method is needed for the second phase and permits the conclusion on the type of redefinition used. This may be not known until the coding phase.

**Algorithm for the search redefined methods**

For a given set of classes, the algorithm below searches for all redefined methods for each class, stores them in appropriate dictionaries and calculates the percentages for each level in the hierarchy. Note that the algorithm parses classes regardless of the fact that they may be organised as a tree hierarchy or as a system, therefore it enables the use of the same algorithm for the calculation of metrics at hierarchy or system level. For this reason, the dictionaries are systematically organised as an n-level entry that corresponds to the n depths of inheritance of the single rooted hierarchy.

---

```

initialise dictionaries
for each class in the branch of the hierarchy
    search at what level the class is situated
    increment the number of classes at the found level
    if no cumulative calculation is required
        store the total number of methods of the class
    else
        store the cumulative number of methods for all ancestor classes of the
        class
    endif
    for each method in the class
        for each superclass of class
            if method signature exists in superclass
                store method name for the class
            endif
        endfor
    endfor
    compute the general redefinition metric for each level of the branch of the
    hierarchy
endfor

```

---

**Algorithm for the search of the type of redefinition used**

For a given set of classes, the algorithm below parses the body of all redefined methods for each class, detects if the methods are either extended or replaced, stores them in appropriate dictionaries and calculates the corresponding percentages for each case at each level in the hierarchy. In this algorithm, the downward parsing task i.e. parsing in subclasses as opposed to ancestor classes, is isolated and can be required by the set-up of a calculation option in the code (see test on parsing direction in the below algorithm). Note that this is not an interactive option as it relates to the calculation algorithm. Downward parsing may be only relevant in the case where a hierarchical structure exists amongst the set of classes assessed e.g. branch of the hierarchy. In such a case, downward parsing is necessary for the construction of the method profiles.

The computation of percentages is done at the end of the algorithm and consists of the direct application of the formula for the considered metric.

---

```

initialise dictionaries
for each class in setOfClasses
  for each method in the class
    if parsing direction = 'Both'
      tempSubclasses ← subclasses in which the method exists
    endif
    booleanExtended ← is current method extended ?
    if tempSubclasses size > 0
      if booleanExtended isFalse
        extendedMethod ← current class
      endif
      for each class in tempSubclasses
        if method is extended
          extendedMethod ← current class
        else
          redefMethod ← current class
        endif
      endfor
    endif
    orig ← find original creator of method
    if orig not in setOfClasses
      if booleanExtended = true
        store method name in dictionary for extended method
      else
        store method name in dictionary for redefined method
      endif
    endif
  endfor
  compute redefinition metric for extended and redefined methods at each level of
  the hierarchy
endfor

```

---

### A.2.7. The method profiles browser

Figure A.8 shows the list of interface classes for the method profile display. It is an instance of the `RedefMethodsBrowser` class that allows the consultation of method profiles for classes. When the user issues such a request, the profile manager object executes the `profile:` method, which in return, initiates the creation of the `RedefMethodsBrowser` instance. The corresponding `MethodsProfile` object is then passed to the browser for display i.e. `setExtendedProfile:` and `setRedefProfile:` methods.

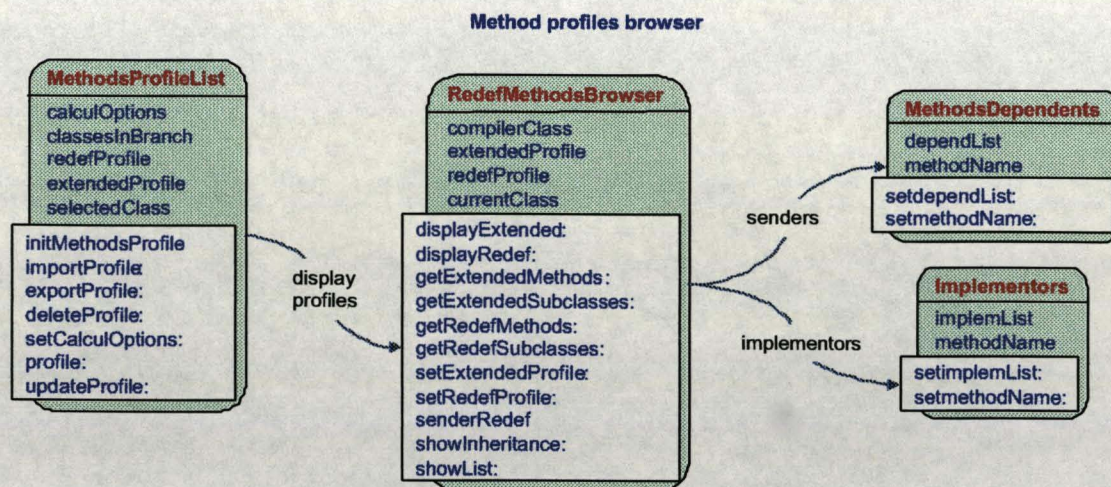


Figure A.8: The method profiles browser design

The class list in the method profile browser can be displayed in two different forms: a flat list or a hierarchical list (respectively realised by the `showList:` and `showInheritance:` methods). This feature facilitates the interpretation of the current branch of the hierarchy when many classes are involved. In addition, for further investigation of one particular method, it is possible for the user to request the list of dependencies with other classes in two ways:

- Search for the senders of the current method (see section 3.3.2): an instance of the `MethodsDependents` class is created. This feature returns a list of classes and the method names that send the current message i.e. method name, thus giving the list of classes dependent on the current one.
- Search for the implementors of the current method (see section 3.3.2): an instance of the `Implementors` class is created. This feature returns the list of classes that implements the current method with their associated depth of inheritance.

### A.3. Remarks on the consequences of the encapsulation mechanism

Suspect uses of inheritance in the Smalltalk class hierarchy are partly due to the absence of an encapsulation mechanism for controlling accessibility of inherited properties. If the proposed redefinition metrics was to be applied on languages where encapsulation mechanisms can be controlled such as C++ and Java, further considerations should be given to the validity and effects of the combination of different property modifiers. In a C++ or a Java application, if methods are declared as public or protected, the metrics would be derived in the same manner as for Smalltalk applications however, when restrained accessibility is applied at class and method level, the use of the redefinition mechanism is inhibited. In Table A.2, in Java, the allowed transitions of method's declaration are shown for a class P, declaring a method m with a modifier x, inherited, redefined and redeclared in a class C with  $C < P$ .

$\Rightarrow$ class C class P	Abstract	Public	Protected	Private	Final
<i>first definition</i> method	✓	✓	✓	✓	✓
Abstract	✓	✓	✓	✓	✓
Public	✗	✓	✗	✗	✗
Protected	✗	✓	✓	✗	✗

Table A.2: Allowed property modifiers for a redefined method in Java

All transitions indicated by a ✗ are forbidden by the Java compiler, therefore, method redefinition cannot take place for those. The case of a method m declared as private has not been included in Table A.2 as, by definition, the accessibility of the method will be restricted to the class only. The issue of the encapsulation mechanism from a measurement point relates to the additional parsing for extracting the necessary design information. In a language such as C++, as only static information is available, a counting strategy of a specific feature may have to take into account subsequent applied property modifiers at method or class level. For example, to compute a number of accumulated methods in a class i.e. including methods from ancestors, the parsing algorithm of the metric collector tool would have to detect any restrictions applied to the methods.