Supporting a model-driven and iterative quality assessment methodology: The MoCQA framework

Benoît Vanderose

Thèse présentée en vue de l'obtention
du titre de Docteur en Sciences

PReCISE Research Centre
Faculty of Computer Science
University of Namur (FUNDP)
Namur, Belgium

December, 2012

©Benoît Vanderose

©Presses universitaires de Namur
Rempart de la Vierge, 13
B - 5000 Namur (Belgique)

Toute reproduction d'un extrait quelconque de ce livre, hors des limites restrictives prévues par la loi, par quelque procédé que ce soit, et notamment par photocopie ou scanner, est strictement interdite pour tous pays.

Imprimé en Belgique
ISBN : 978-2-87037 -780-2
Dépôt légal: D / 2012 / 1881 / 42

Doctoral Committee

Prof. Vincent Englebert (Chair)
Faculty of Computer Science
University of Namur

Prof. Naji Habra (Advisor)
Faculty of Computer Science
University of Namur

Prof. Anthony Cleve (Internal Reviewer)
Faculty of Computer Science
University of Namur

Prof. Tom Mens (External Reviewer)
Faculty of Science
University of Mons

Prof. Lionel Briand (External Reviewer)
Faculty of Science, Technology and Communication
University of Luxembourg

Abstract

Software Quality has been a major and focal concern of Software Engineering since its infancy. Despite the proficiency of research addressing quality, quantitative quality assessment methods remain mostly inefficient in industrial contexts. Besides, they are mainly used to control and not to guide the developers, decreasing drastically their potential. As a result, although the field itself is mature and provides a wealth of knowledge, the practical quality assessment of software has still not reached a state where it may be performed satisfactorily.

In this research, we propose a framework that supports model-driven and iterative quality assessment in order to help leverage the potential of quantitative assessment and integrate it into the software development process in a more coherent way. This Model-Centric Quality Assessment (MoCQA) framework defines a goal-driven assessment methodology that allows the exploitation of operational customised quality assessment models (or MoCQA models) through a dedicated quality assessment metamodel. The use of a quality assessment metamodel guarantees the integration of heterogeneous quality models and software measurement methods in MoCQA models and let these models adopt an ecosystemic viewpoint on software quality. Besides, the methodology relies extensively on the involvement of stakeholders and let them steadily construct a common mental model of the quality aspects at stakes for a given development project.

Through these mechanisms, the framework intends to provide the necessary support for the integration of multiple quantitative quality assessment methods (both existing ones and customised ones) into any type of development and maintenance life-cycles in a meaningful, self-aware and flexible way.

Résumé

La Qualité Logicielle est un défi majeur et capital du Génie Logiciel depuis ses débuts. Malgré la profusion de travaux de recherche abordant la qualité, les méthodes quantitatives d'évaluation de la qualité restent majoritairement inefficaces dans un contexte industriel. De plus, elles sont principalement utilisées pour contrôler les développeurs au lieu de les guider, diminuant de ce fait leur potentiel. En conséquence, bien qu'étant un domaine mature ayant accumulé de nombreuses connaissances, l'évaluation de la qualité des logiciels n'a toujours pas atteint un état lui permettant d'être exécutée de manière satisfaisante.

Dans cette recherche, nous proposons un cadre de référence supportant une évaluation de la qualité guidée par les modèles, itérative et incrémentale de sorte à tirer avantage du potentiel de l'évaluation quantitative et à l'intégrer de manière plus cohérente dans le processus de développement. Ce cadre de référence MoCQA (Model-Centric Quality Assessment) définit une méthodologie d'évaluation guidée par les buts qui permet l'exploitation de modèles de l'évaluation de la qualité personnalisés et opérationnels (ou modèles MoCQA), grâce à un métamodèle de l'évaluation de la qualité. Ce métamodèle garantit l'intégration de modèles de qualité et de méthodes de mesure hétérogènes au sein des modèles MoCQA et permet à ces modèles d'adopter un point de vue écosystémique de la qualité logicielle. De plus, la méthodologie s'appuie sur l'implication des acteurs afin qu'ils puissent se construire peu à peu un modèle mental commun des aspects de qualité primordiaux pour un projet de développement donné.

A l'aide de ces mécanismes, le cadre de référence se veut capable de fournir le support nécessaire à l'intégration de multiples méthodes d'évaluation (à la fois existantes et personnalisées) au sein de n'importe quel cycle de vie de développement ou de maintenance, et ce d'une manière significative, réflexive et flexible.

Acknowledgements

Although carrying out doctoral research may appear as a lonely road at times, I am fortunate enough to have met many people willing to show me the way all along this road. I would therefore like to thank all the people who contributed directly or indirectly to this research.

First of all, I would like to express my gratitude to my advisor, Prof. Naji Habra, who guided me through all the challenges I have encountered during the course of this research. Beyond his precious advices on my work, his mentoring skills really helped me address these challenges in a more serene and focused state of mind than I would ever have been able to reach by myself. Besides, I consider myself very fortunate to have learned the fine art of negotiation - sometimes the hard way - from such a master.

I would also like to thank the members of my doctoral committee, Prof. Lionel Briand, Prof. Anthony Cleve, Prof. Vincent Englebert and Prof. Tom Mens, for their valuable advices and encouraging feedback. Meeting them was an honour and discussing my research with them has been a very enriching experience.

During the course of this research, many people contributed to help me apply my work outside the walls of the Faculty. I would like to thank Samuel Hanoteau for his dedication to the application of the MoCQA framework within his work environment and for the resulting collaboration. Similarly, I wish to express my gratitude to the members of the CETIC, and more specifically, to Christophe Ponsard for believing in the relevance of the MoCQA framework and letting me apply the framework in the various projects in which I've been involved. Finally, I would like to thank Claude Dekrom for allowing me to field-test the MoCQA framework in his work environment.

During the past years, many students contributed to the testing of a more than experimental MoCQA framework. I therefore thank all of them and especially Lorent Lempereur for his valuable contribution to the QuaTALOG project.

The effort carried out during the past years would have been unbearable without a welcoming work environment. I therefore thank my colleagues and friends Abdelkader, Alain, Hejer and Flora for making the workplace a nice place to

spend all this time. Thanks to them, what could have been a simple office has become a vibrant and lively place where many good times were - and still are - shared. More generally, I would like to thank all the colleagues who contribute to make this Faculty a place where hallways witness both hight-level intellectual debates and (almost) appalling humour. I also thank the many colleagues that contributed to this research by sharing their knowledge, dispensing useful feedback and providing me with new and interesting ideas to explore.

On a more personal note, I would like to thank all the people who shape and colour my every day life. First and foremost, I would like to thank the members of my family for their love, trust and support, which provided me with the outline on which I would be able grow as a person. Besides, in the light of the past few years, I would like to thank my parents for teaching me what has been the most essential principle throughout my work: always look at what has already been achieved and not at the amount of work that is still awaiting.

I am also very grateful to be able to rely on close friends who have been around for quite a long time now. I especially owe Johann, Sandy, Stéphane (a.k.a. Vitou) and Ravi for dissolving many of my doubts, listening to my existential questions and for participating in so many crucial debates on the meaning of life, love and...stuff. Besides, I am very grateful to Ravi for always leading the way towards the unknown territories I enter two years after him and being kind enough to provide me with his invaluable insights afterwards. I also thank them all, as well as all my other friends, for the good times we share, for the laughs and joy and for the shades of colour they add to the overall picture.

Finally, I would like to thank my beloved wife, Anne, for her patience, understanding, support and for never asking me to be someone other than my silly self. Her presence and unconditional love is the canvas that makes everything else possible.

Contents

Contents **ix**

List of Tables **xv**

List of Figures **xvii**

I Research Context **7**

1 Software Quality **9**

1.1 Quality models . . . 10

1.1.1 First influential researches . . . 10

1.1.2 ISO/IEC 9126 Quality Model and variations . . . 13

1.1.3 Domain-specific quality models . . . 14

1.1.4 Other quality frameworks . . . 16

1.2 Software Measurement . . . 17

1.2.1 Fundamentals of software measurement . . . 17

1.2.2 Software Measures . . . 19

1.2.3 Implementation of measurement programs . . . 23

1.3 Software process improvement . . . 27

1.4 Quality modelling . . . 31

1.4.1 GenMETRIC and SMML . . . 31

1.4.2 QMM and Quamoco . . . 32

1.4.3 Other quality metamodels . . . 33

2 Research issues **35**

2.1 Issues related to quality models . . . 35

2.1.1 Complexity of the operationalisation . . . 36

2.1.2 Confusion between quality models and quality modelling . . . 37

2.2 Issues related to software measurement . . . 38

2.2.1 Conceptual misconception pertaining to measurement . . . 38

2.2.2 Lack of empirical validation 39
2.2.3 Complexity of measurement programs implementation . . . 39
2.3 Issues related to the integration of quality assessment into the software development 40
2.3.1 Spread of measurement methods 40
2.3.2 Problematic role of quality assessment 40
2.3.3 Impact of model-driven engineering 41
2.3.4 Impact of software ecosystems 41
2.3.5 Organisational issues regarding quality assessment 42
2.3.6 Cost and effort of quality assessment/improvement 42

3 Conceptualisation of the domain 45
3.1 Terminology 45
3.2 Ontology 51
3.2.1 Purpose 52
3.2.2 Building the ontology 52
3.2.3 Evaluation and documentation 52
3.2.4 Software Quality ontology 53

II Model-Centric Quality Assessment 57

4 Overview of the approach 59
4.1 Objectives of the approach 59
4.2 Founding principles 63
4.2.1 Constructivism 63
4.2.2 Iterative / incremental life-cycle 64
4.2.3 Involvement of the stakeholders 65
4.2.4 Goal-Driven definition of measures 66
4.2.5 Ecosystemic viewpoint 66
4.2.6 Definitional and analytic approaches integration 67
4.2.7 Reusability 68
4.2.8 Domain-specific languages and expressiveness 68
4.2.9 Human aspect of software quality 69
4.3 The MoCQA framework 71
4.3.1 MoCQA models 71
4.3.2 Model-Centric Quality Assessment methodology 72
4.4 Structure 79

5 MoCQA models 81
5.1 MoCQA models and Meta-Object Facility 82
5.2 Quality assessment metamodel 84
5.2.1 Project package 87

5.2.2 Measurement package 102
5.2.3 Assessment package 110
5.3 Designing the MoCQA model 124
5.3.1 Components instantiation 125
5.3.2 Structural coherence 125

6 Step 1: Acquisition 129
6.1 Overview 129
6.1.1 Activities 129
6.1.2 Formalisation 132
6.2 MoCQA model design in practice 132
6.2.1 Customising existing quality models 133
6.2.2 Developing analysis grids 134
6.2.3 Tool support 135
6.2.4 Complementarity with scenario-based analysis 136
6.2.5 Complementarity with Requirements Engineering 136

7 Step 3: Measurement Plan 139
7.1 Overview 139
7.1.1 Activities 139
7.2 MoCQA model transformations 141
7.3 Operationalisation challenges 143
7.3.1 Formalisation of the operationalisation 145
7.4 Preparing data collection 146
7.4.1 Data Model 146

8 Step 5: Exploitation 149
8.1 Overview 149
8.1.1 Activities 149
8.2 Quality Profiling 150
8.2.1 Interpreting quality indicators 151
8.2.2 Supporting root-cause analysis 151
8.2.3 Exploiting MoCQA models during software evolution . . . 152
8.2.4 Exploiting MoCQA models at early stages of the development 155
8.3 Reviewing MoCQA models 155
8.3.1 Content integrity 155
8.3.2 MoCQA-related indicators 158
8.3.3 Illustration 162

9 Tool support 165
9.1 Tool-related challenges of model-driven quality assessment 165
9.2 Model-driven tools and MoCQA framework 166

9.2.1 Exploitation of model constraints 167
9.2.2 Exploitation of model transformation languages 168
9.2.3 Co-evolution of models and collaborative modelling 168
9.3 Dedicated and integrated tool support 169
9.3.1 XML-based Operational Customised Quality Assessment Model 169
9.3.2 MoCQA Utilities on the Go (MUG) 170
9.3.3 OCQAM editor 172
9.3.4 QuaTALOG 172
9.3.5 Towards and integrated tool support 175

III Validation of the approach **179**

10 Validation process **181**
10.1 Research questions 181
10.2 Challenges 183

11 Operationalisation of quality models **189**
11.1 Objectives 189
11.2 ISO/IEC 9621 quality model 190
11.2.1 Overview 190
11.2.2 Instantiation 191
11.2.3 Results 192
11.3 QualOSS documentation availability model 193
11.3.1 Overview 193
11.3.2 Metamodel Instantiation 195
11.3.3 Results 196
11.4 Discussion 197
11.5 Threat to validity 198

12 Quality of software architecture **199**
12.1 Context 199
12.2 Objectives 200
12.3 Architecture trade-off analysis method 200
12.4 Architecture analysis method with MoCQA 201
12.4.1 Utility trees and MoCQA models 202
12.4.2 First proposal of quantitative assessment 203
12.4.3 Second proposal of quantitative assessment 205
12.4.4 Third proposal of quantitative assessment 207
12.5 Results 210
12.5.1 Support for utility tree processing 210
12.5.2 Support for quality traceability 210

12.5.3 Support for architecture refactoring decisions 211
12.5.4 Support for architecture design decisions 211
12.5.5 Flexibility of the approach 211
12.6 Discussion 212
12.7 Threat to validity 212

13 Empirical studies 215
13.1 Preliminary study 215
13.1.1 Context and objectives 215
13.1.2 Description 216
13.1.3 Results 216
13.1.4 Discussion and threat to validity 216
13.2 Preliminary study: Quality of OSS 217
13.2.1 Context and objectives 217
13.2.2 Description 217
13.2.3 Results 218
13.2.4 Discussion and threat to validity 218
13.3 Support for software maintenance and evolution 219
13.3.1 Planning of the study 219
13.3.2 Design of the study 220
13.3.3 Experimental study 226
13.3.4 Results 229
13.3.5 Discussion 231
13.3.6 Threat to validity 231

14 Supporting certification 233
14.1 Context 233
14.2 Objectives 234
14.3 Description 235
14.3.1 RTCA DO-178b 235
14.3.2 Variability and Software Product Lines 236
14.3.3 Applying the MoCQA framework 236
14.3.4 Towards selective certification 237
14.4 Results 238
14.5 Discussion 239
14.6 Threat to validity 239

15 Quality Assurance 241
15.1 Context 241
15.2 Objectives 242
15.3 Description 242
15.3.1 First quality assessment cycle 242

15.3.2 Continuation of the quality assessment life-cycle 246
15.4 Results 246
15.5 Discussion 247
15.5.1 Impact of quality indicators 249
15.5.2 Human aspects 249
15.5.3 Stakeholder classification 250
15.5.4 Target of the assessment 251
15.5.5 Availability of results 251
15.5.6 Support from the management 252
15.6 Threat to validity 252

IV Closing comments 255

16 Discussion 257
16.1 Contribution 257
16.2 Review 259
16.3 Limitations 261
16.4 Perspectives 263

Conclusion 271

Bibliography 275

Index 291

List of Tables

5.1 Attributes characterising an artefact type construct 91
5.2 Relationships involving artefact types 92
5.3 Attributes characterising a behaviour type construct 95
5.4 Relationships involving behaviour types 95
5.5 Attributes characterising a derivation type construct 98
5.6 Relationships involving derivation types 99
5.7 Attributes characterising a base attribute construct 104
5.8 Relationships involving base attributes 105
5.9 Attributes characterising a method construct 107
5.10 Relationships involving methods 107
5.11 Attributes characterising a derived attribute construct 109
5.12 Relationships involving derived attributes 110
5.13 Attributes characterising a function construct 111
5.14 Relationships involving functions 111
5.15 Attributes characterising a quality issue construct 114
5.16 Relationships involving quality issues 115
5.17 Attributes characterising an assessment model construct 116
5.18 Relationships involving assessment models 117
5.19 Attributes characterising a quality indicator construct 119
5.20 Relationships involving quality indicators constructs 120
5.21 Attributes characterising an interpretation rule construct 123
5.22 Relationships involving interpretation rules 123
5.23 Mandatory attributes 126

13.1 Measurement values from the first version of the MoCQA model . . . 226
13.2 Measurement values from the improved version of the MoCQA model 227
13.3 Global completeness indicators 229

List of Figures

1.1 McCall's quality model 10
1.2 Boehm's quality model 11
1.3 ISO/IEC 9126 internal and external characteristics 13
1.4 ISO/IEC 9126 quality in use characteristics 13
1.5 SQuaRE's product quality characteristics 15
1.6 SQuaRE's quality in use 16
1.7 Quality in conceptual modelling 17
1.8 Relationship between real and formal worlds through measurement. . 18
1.9 GQM/MEDEA conceptual model 25
1.10 MOSME data model 26
1.11 MIM conceptual model 27
1.12 CMMI maturity levels 30
1.13 GenMETRIC underlying metamodel 32
1.14 Quamoco metamodel for specifying and evaluating software quality . . 33

3.1 The software quality assessment ontology 53

4.1 Integration of quality models and measurement/estimation methods . 73
4.2 The MoCQA methodology 75

5.1 The four layers of modelling 83
5.2 Multi-level hierarchy for the approach 84
5.3 Simplified view of the MoCQA quality assessment metamodel 85
5.4 Process view of the ISO/IEC 15939 standard 86
5.5 Project package of the quality assessment metamodel 87
5.6 Project level of the software quality ontology 89
5.7 Two basic artefact types 92
5.8 An artefact type with children artefact types 93
5.9 Artefact types with a reduced entity population 93
5.10 Artefact types with a very focused entity population 94

5.11 Example of a basic behaviour type 96
5.12 Example of more specific behaviour types 97
5.13 Misleading relationship between 2 measurable entity types 97
5.14 Example of derivation type 100
5.15 Example of automated derivation type 101
5.16 Measurement package of the quality assessment metamodel 102
5.17 Measurement level of the software quality ontology 103
5.18 Example of base attributes 106
5.19 Example of measurement/estimation methods 109
5.20 Example of derived attributes and functions 112
5.21 Assessment package of the quality assessment metamodel 113
5.22 Assessment-level of the software quality ontology 113
5.23 Example of quality issues and assessment models 118
5.24 Example of quality indicators 121
5.25 A complete (yet simple) example of MoCQA model 122

7.1 Introduction of a collection of entity type 142
7.2 Removal of a derivation type 143
7.3 The MoCQA data model 146

8.1 A simplified MoCQA model applied to co-evolution 153
8.2 Example MoCQA model 161

9.1 MUG user interface 171
9.2 QuaTALOG user interface 174

11.1 ISOQM explicited metamodel 190
11.2 Related quality assessment metamodel concepts 190
11.3 QualOSS robustness and evolvability quality model 193
11.4 QualOSS documentation availability model 194
11.5 QDAM explicited metamodel 194
11.6 Quality assessment metamodel concepts 194
11.7 QDAM targeted artefacts 195

12.1 BCS utility tree 202
12.2 BCS utility tree expressed with MoCQA constructs 203
12.3 Hardware (deployment) view of the BCS 204
12.4 Evaluation based on behaviour types 205
12.5 Evaluation based on code-related artefact types 206
12.6 Evaluation based on design-related artefact types 209

13.1 Completeness quality factor decomposition 221
13.2 Basic MoCQA model (measurement and project components) 223

13.3 Refined MoCQA model (measurement and project components) 224
13.4 Boxplots for each of the 4 measures 228
13.5 Comparison between global indicators 229

14.1 Traceability requirements 235
14.2 Test preparation and test execution 236
14.3 Modelling the traceability of test cases 237

15.1 Example of quality issues expressed during the case study 243
15.2 Example of MoCQA model designed during the case study 244

16.1 Collaborative and iterative validation/refinement methodology 265

Introduction

Quality has been a major and focal concern of software engineering since its infancy. According to [Peters and Pedrycz, 1998], producing *reliable* software may even be regarded as the only objective of software engineers. However, within the context of a constantly evolving field like software engineering and with the steadily increasing level of complexity of software, what software quality means has become increasingly delicate to define. As new fields and paradigms of software engineering have been appearing, quality concerns have been dispatched into several different and more or less independent subdomains. Quality assessment has therefore become a concern in every field of software engineering (from requirements engineering to design and coding). As a result, several quality assessment approaches (i.e., quality models, software measurement methods, etc.) have been proposed for the past three decades. This increasing number or methods contributed to make Software Quality a vast and complex field of Software Engineering.

Problem Statement

Despite the proficiency of research works addressing quality, the main observation remains the overall misguided and/or inefficient use of measures in industry, leading to costly [Fenton and Neil, 1999] or useless measurement plans. Some surveys (notably [Kasunic, 2006]) also show that measurement tends to appeal more to the management than it does to the development team. This reflects the fact that quantitative approaches are mainly used to control and not to guide the developers, decreasing drastically the potential of quantitative quality assessment.

A notable curb to the adoption of quantitative approaches as an integrated tool of the development is the vast amount of different proposed quantitative measurement methods and the fact that they have been proposed for almost every level of abstraction and type of products. This wealth of available methods makes it complex to sort out the more suitable ones and use them correctly within a development team. The same is true for specific quality models.

Another hindrance to the general adoption of measurement is the frequent lack of clarity about what metrics actually measure and the quality concepts they reflect. This lack of clarity appears both on a structural level (due notably to a general lack of experimental validation of metrics [Riguzzi, 1996]) and in the way measures are used (due to the lack of awareness regarding what goal the measurement pursues). Quantitative approaches are thus promising but require frameworks that supply the theoretical support needed to clarify the intent of use of the measures.

The constant evolution of Software Engineering induces many changes in the way software is perceived and envisioned. As explained in [Schmidt, 2006], the apparition of model-driven engineering, for instance, introduced a whole new point of view on what software is, moving the focus on the intermediary artefacts involved in software development and putting the focus on the models over the code. In the context of quality assessment, model-driven engineering introduces new challenges and the need for more flexible quality assessment frameworks allowing to take into account multiple levels of abstraction as well as the relationships between artefacts from these different levels [Mohagheghi and Dehlen, 2008]. Moreover, the paradigm is still vaguely defined and encompasses several different realities and practices [Vignaga, 2007], increasing the need for more flexible and adaptive quality assessment methods.

More recently, the increasing attention paid to the notion of software ecosystem [Lungu, 2009] led to a new shift in the way software is perceived. The notion of software ecosystem forces developers to consider additional factors (such as social aspects [Mens and Goeminne, 2011]) that influence the development of software and, therefore, the way it may be assessed.

Finally, quality assessment possesses an intrinsic human-related aspect [Westfall and Road, 2005] that cannot be ignored. Measurement methods may evolve from the technical point of view and become very accurate at reporting defects of a software system. However, the way the evaluation is perceived by the development team remains a crucial factor in the successful exploitation of the measurement values collected.

As a result, the main problem that still pertains to Software Quality is the fact that, although the field itself is mature and provides a wealth of knowledge, the practical quality assessment of software still has not reached a state where it may be performed satisfactorily (i.e., in such a way it would fulfil all involved actors' expectations). The research work described in this dissertation intends to bridge this gap between the theoretical richness and the practical misuses of quality assessment. It also helps leverage the potential of quantitative software quality assessment and integrate quality assessment into the software development process in a more coherent way.

Research questions

In order to tackle this problem, the following research questions have been formulated in order to guide our research work.

[**RQ1**] How can we provide methodological elements to support a flexible and meaningful quality assessment process?

In order to bridge the gap between the Software Quality body of knowledge and the way it is exploited, a methodology that helps streamline the quality assessment process is required. Regarding Software Engineering itself, many methodological refinements have been proposed to improve the way software development is carried out (i.e., iterative/incremental methods, model-driven engineering, etc.). Answering this question thus requires to consider which methodological elements of software engineering are applicable to quality assessment in order to facilitate its execution.

[**RQ2**] How can we formalise the quality assessment process so that the heterogeneous expectations of all involved actors are understood by each other?

In order to address human-related aspects, a key aspect of quality assessment is providing better ways to formalise its elements. Better formalisms (e.g., better syntax, efficient graphical notations, etc.) should be provided in order to improve the communication regarding actors' expectations among them.

[**RQ3**] What practical techniques may support the effective integration of quality assessment into the software development and maintenance processes?

Provided with adequate methodologies, the actors involved in software development still need concrete mechanisms, tools and formalisms to support the methodology. This question addresses the definition of such techniques (i.e., models, textual notations, tool-support, etc.) that are both applicable in the methodological context defined and still applicable to quality assessment.

[**RQ4**] How can we ensure the coherent integration of various quality assessment methods within the same environment?

In addition to the requirement of adequate techniques to support the quality assessment process, the challenge of reusing existing quality assessment methods remains. Providing a formalised way (e.g., ontology, metamodel, etc.) to integrate heterogeneous methods within the process is therefore another crucial aspect of the successful execution of quality assessment.

[**RQ5**] How can quality assessment adapt to the evolution of the way software is defined and perceived?

As explained before, software is not perceived as a black-box monolithic piece of code anymore. This means that quality assessment methodologies have to integrate new ways of considering software (the same way model-driven engineering

or software ecosystemic approaches do) in order to address it in a coherent way. This question therefore supposes to redefine the abstraction level at which quality assessment is envisioned.

Contribution

Taking all the above in consideration, the main contribution of this research work is:

> The achievement of a *framework* that provides the necessary support for integrating quantitative quality assessment methods (both existing and customised ones) into any type of development or maintenance life-cycle, in a meaningful (i.e., useful for all stakeholders), self-aware (i.e., allowing a critical review of the process) and flexible (i.e., easily adaptable to any type of environment) way.

This Model-Centric Quality Assessment (MoCQA) framework relies on techniques inherited from various fields of software engineering, such as (meta)modelling. It relies on a quality-related ontological support to provide quality assurance teams with a structured and flexible set of procedures that support the implementation of a quality assessment plan throughout the development life-cycle. The approach implemented in the framework intends to be a transversal take on quality that places quality assessment at the project level. It therefore provides a global and integrated view on quality concerns for all involved stakeholders. Besides, it addresses both product-oriented and process-oriented quality assessment within the same context. Finally, it emphasises the importance of communication about quality aspects and proposes several techniques to ensure the efficiency of this communication.

Structure

This dissertation is organised as follows.

Part I addresses the context of our research work. Chapter 1 presents related efforts that have been used as a foundation for the approach introduced in this dissertation or are meant to complement it. Chapter 2 details the shortcomings and various issues inherent to Software Quality. Chapter 3 concludes this part of the dissertation with a conceptualisation of the research context designed to support the construction of our approach.

Part II details the proposed approach, introduces the techniques the framework relies on and the methodology provided to use them efficiently. Chapter 4 provides an overview of the approach and details its theoretical foundations. Chapters 5

to 9 describe practical aspects of the theoretical approach in order to provide a comprehensive overview of our framework.

Part III addresses the validation of the framework. Chapter 10 provides an overview of the validation process that was followed during this research work. It defines a set of criteria to help structure the validation of the framework. Chapters 11 to 15 then report case studies that were carried out to test the framework and to contribute to show that the framework satisfies the previously defined criteria.

Finally, **Part** IV provides closing comments, as well as a list of future efforts considered to achieve the long-term objectives of the approach.

Part I

Research Context

Chapter 1

Software Quality

As explained in the introduction, Software Quality has been a fundamental concern for software engineers for the past three decades. As a consequence, the issues regarding how to define software quality, how to evaluate the overall quality of software products and how to grant a satisfactory level of quality have been (and still are) abundantly investigated. This chapter intends to provide a transversal overview of the current Software Quality body of knowledge and to illustrate the vast scope of Software Quality as a field.

The research works presented in this chapter have been regrouped in four subfields of study. Section 1.1 provides an overview of the declarative approaches to quality assessment (i.e., quality models and quality frameworks). These approaches intend to define more accurately what quality is, as well as they aims to structure and formalise this definition. In Section 1.2, analytical approaches to quality assessment (i.e., software measurement) are described. These approaches draw on the metrology body of knowledge in order to apply its concepts to software engineering. They intend to allow a quantitative characterisation of software products. Section 1.3 provides an overview of the quality improvement approaches found in the Software Quality literature. These approaches build on the notion that the overall quality of a software product results from the quality of the processes used during its development life-cycle. They therefore seek to evaluate and improve these processes. Finally, Section 1.4 addresses a more recent trend in Software Quality, that is, the use of metamodeling techniques and model-driven principles to support the definition and management of software quality.

For each of these sections, the research efforts introduced are organised chronologically inasmuch as possible and have been selected in order to provide a hint at the variety and the profusion of quality assessment methods available to the researchers and developers.

1.1 Quality models

1.1.1 First influential researches

The idea of structuring software quality into smaller and easier-to-assess quality factors is not new. The core of hierarchical quality models relies on this principle in order to provide a better characterisation of software quality: software quality is organised into several pillars that are further refined into quantitatively assessable factors for which measures are defined. This approach to quality assessment may be regarded as reminiscent of the divide and conquer algorithm design method.

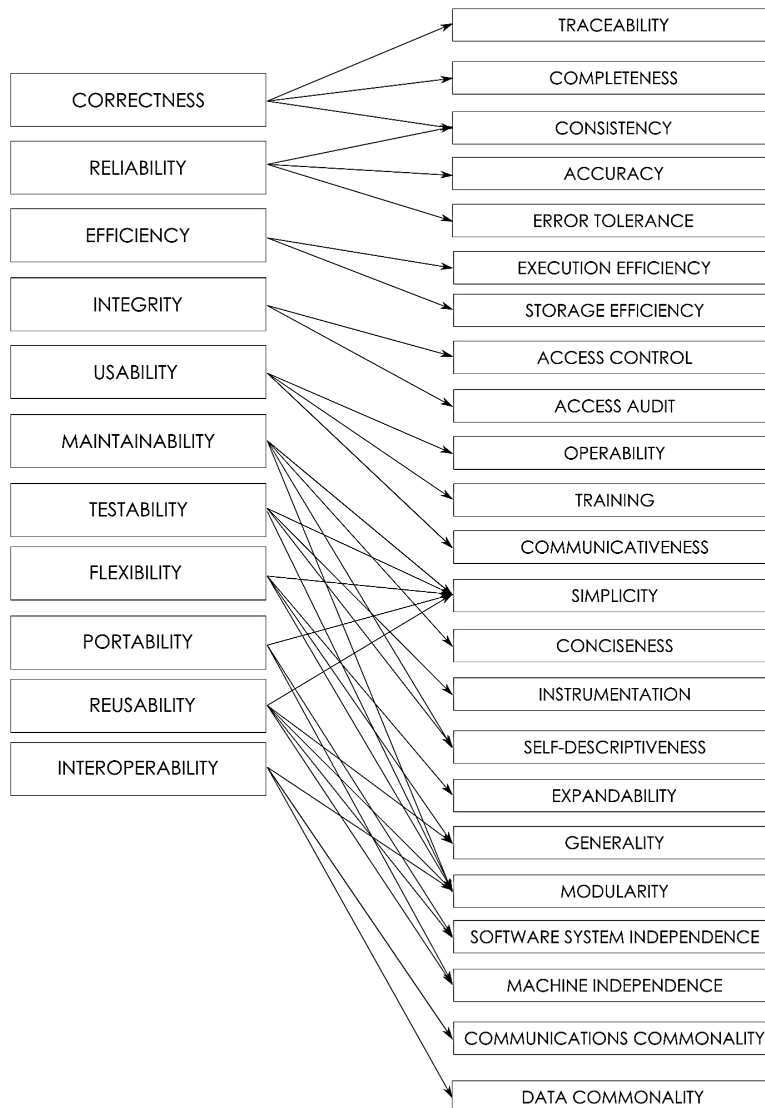

Figure 1.1: McCall's quality model

McCall's quality model

The earliest proposal of hierarchical quality model can be found in [Mccall et al., 1977]. McCall's model introduces eleven quality factors related to three different aspects of software quality: product revision (i.e., the ability to undergo changes), product transition (i.e., the ability to adapt to new environments) and product operations (i.e., its functional aspects). These factors (which are equivalent to ISO/IEC 9126's external characteristics) are refined into twenty-three criteria (which are internal to the product). Those criteria are then associated with actual metrics (percentage of positive answers to a list of questions associated to the criterion). Figure 1.1, adapted from [Pfleeger, 1998], shows the relationships between factors and criteria. Regarding McCall's model, [Ortega et al., 2003] states:

> One of the major contributions of the McCall model is the relationship created between quality characteristics and metrics, although there has been criticism that not all metrics are objective. One aspect not considered directly by this model was the functionality of the software product.

Boehm's quality model

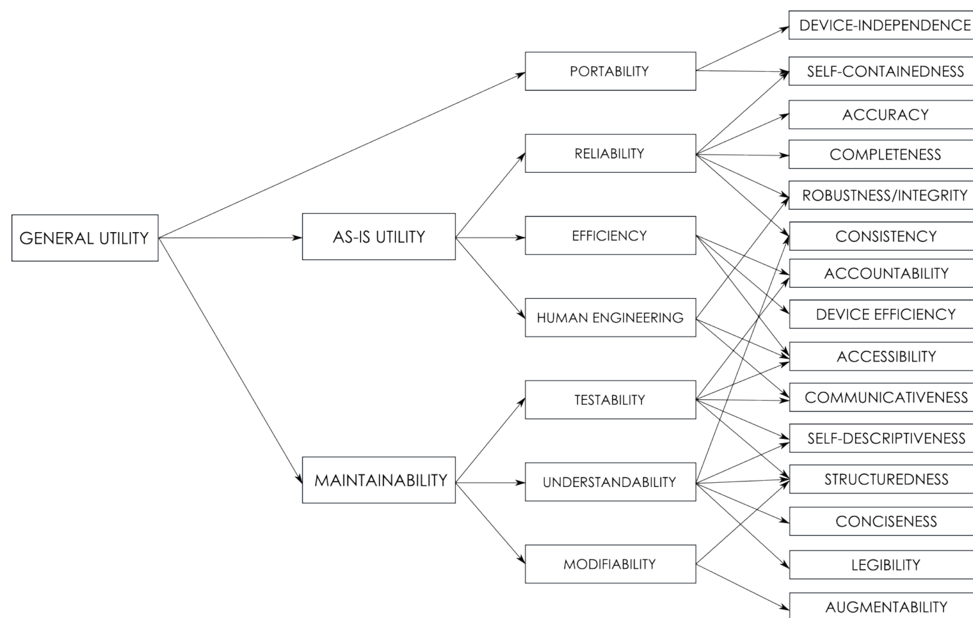

Figure 1.2: Boehm's quality model

Boehm's quality model, defined in [Boehm, 1981], is essentially a refinement of McCall's. It adds some characteristics to the latter and increases the emphasis

on maintainability of software products. The introduction of an assessment of the utility of the product is worth mentioning. The proposal only includes the quality factors hierarchy (shown in Figure 1.2) but no support for the evaluation of the factors is provided. However, the metrics are part of the decomposition since the "layers" of the quality model are defined as: high-level characteristics, primitive characteristics and metrics. According to [Ortega et al., 2003]:

> Boehm's model is similar to the McCall model in that it represents a hierarchical structure of characteristics, each of which contributes to total quality. Boehm's notion includes users needs, as McCall's does; however, it also adds the hardware yield characteristics not encountered in the McCall model.

FURPS

Introduced in [Grady and Caswell, 1987], the FURPS model decomposes the characteristics into functional (Functionality) and non-functional (Usability, Reliability, Performance and Supportability) ones. The use of FURPS consists in setting priorities (i.e., defining which characteristic is more important if one of them can be increased at the expense of another) and then defining the quality attributes that are related to the characteristics and that can be measured. A refined version of the model named FURPS+ [Grady, 1992] exists and introduces more constraints on various aspects of the software development process.

Dromey's quality model

Another influential proposal is Dromey's model [Dromey, 1995; 1996]. The proposed framework has been designed to help build an operational quality model. The framework distinguishes three separates categories of quality models, depending on the product that is assessed and its place in the software development (Requirement Determination, Design, Implementation). It also takes into account the fact that some high-level attributes (e.g., reliability or maintainability) can not be 'built into the software product' but only provided by the identification of clearly defined properties that have to be fulfilled in order to provide the desired high-level attribute. Regarding this aspect, [Ortega et al., 2003] specifies that:

> Dromey's model seeks to increase understanding of the relationship between the attributes (characteristics) and the sub-attributes (sub-characteristics) of quality. It also attempts to pinpoint the properties of the software product that affect the attributes of quality.

1.1.2 ISO/IEC 9126 Quality Model and variations

Although it was first introduced in 1991 (and therefore before Dromey's model), the ISO/IEC 9126 has undergone several modifications in order to become the stable standard version of 2001 that has been used for many years [ISO/IEC, 2001a]. This model provides a hierarchical structure divided into 4 layers : quality characteristics, sub-characteristics, attributes and metrics. It distinguishes internal characteristics (static aspects of the software product) from external characteristics (dynamic aspects of the software product) and also considers "quality in use" characteristics, which are the quality characteristics considered from the end-user point of view.

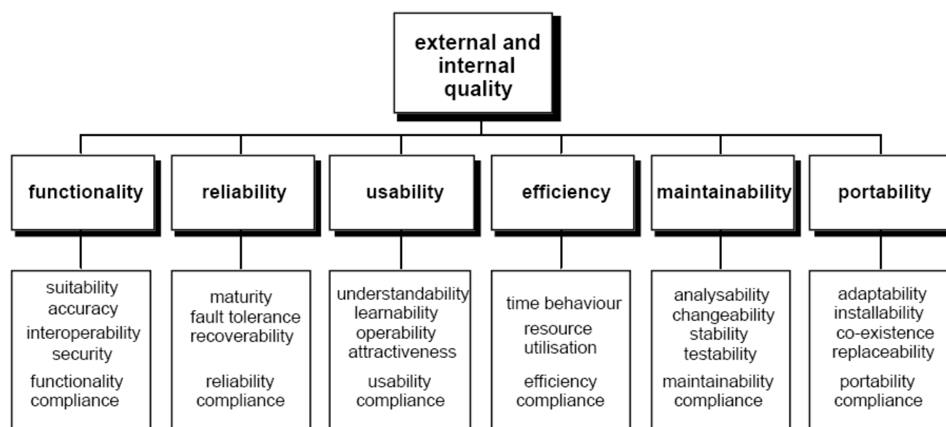

Figure 1.3: ISO/IEC 9126 internal and external characteristics

The ISO/IEC 9126 quality model relies on a definition of software product that is very broad and encompasses various elements (i.e., computer programs, procedures, and possibly associated documentation and data). It is decomposed in six quality characteristics that are further refined in sub-characteristics, as shown in Figure 1.3. It also structures in use quality according to four characteristics as shown in Figure 1.4.

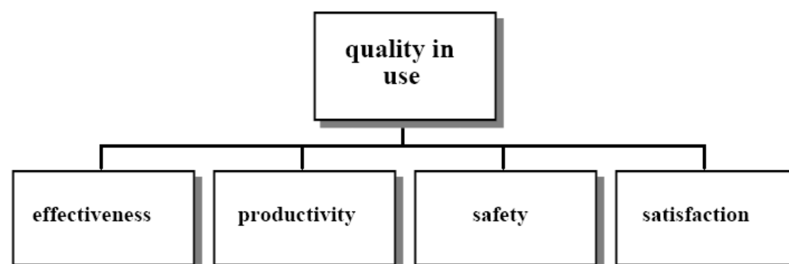

Figure 1.4: ISO/IEC 9126 quality in use characteristics

The standard also defines number of metrics (discussed in Section 1.2) to evaluate the quality characteristics, via specific attributes. However, one notable feature of the ISO/IEC quality model is that it is very generic, due probably to its international standard status. The set of metrics proposed to evaluate the software product are consequently vague. According to [Ortega et al., 2003]:

> One of the advantages of this model is that it identifies the internal characteristics and external quality characteristics of a software product. However, at the same time it has the disadvantage of not showing very clearly how these aspects can be measured.

An aspect of the ISO/IEC 9126 quality model directly related to the previous one is that it offers the possibility to be tailored and adapted to different domains. It is thus often used as a basis for the development of various quality models addressing more specialised topics such as software architecture assessment [Losavio et al., 2001; 2004], test specifications [Zeiss et al., 2007], B2B applications [Behkamal et al., 2009] or commercial off-the-shelf (COTS) solutions [Torchiano et al., 2002] to name a few. This process of adaptation and tailoring is never trivial and requires a vast effort in order to be accomplished thoroughly.

In recent years, the ISO/IEC 9126 quality model has been improved and integrated into the Software product Quality Requirements and Evaluation (SQuaRE) standards [ISO/IEC, 2005b]. This second generation of quality standards aim to satisfy *"the evolving needs of users through an improved and unified set of normative documents covering three complementary quality processes: requirements specification, measurement and evaluation* [Suryn et al., 2003]. The SQuaRE standards represent an attempt to align the various quality-related existing standards (i.e., ISO/IEC 91xx, 14xx and 15xx) in a harmonised structure. They also reorganise the quality standards (as a set of 14 documents). These documents introduce a new general reference model, detailed guides, a standard on Measurement Primitives, a standard on Quality Requirements and a series of examples designed to provide better guidance [Suryn et al., 2003]. Finally, the SQuaRE standards provide a better integration of the Measurement Information Model (discussed in Section 1.2).

As a result, SQuaRE's quality model improves the ISO/IEC 9126 quality model through a revision of the nomenclature of quality characteristics (e.g., "functionality" becomes "functional suitability"), a structural reorganisation of the subcharacteristics and the addition of subcharacteristics (e.g., functional completeness, compatibility, etc.) as shown in Figure 1.5 and 1.6.

1.1.3 Domain-specific quality models

To conclude this review of influential hierarchical quality model proposals, it is worth mentioning some examples of other quality models that, while not deriv-

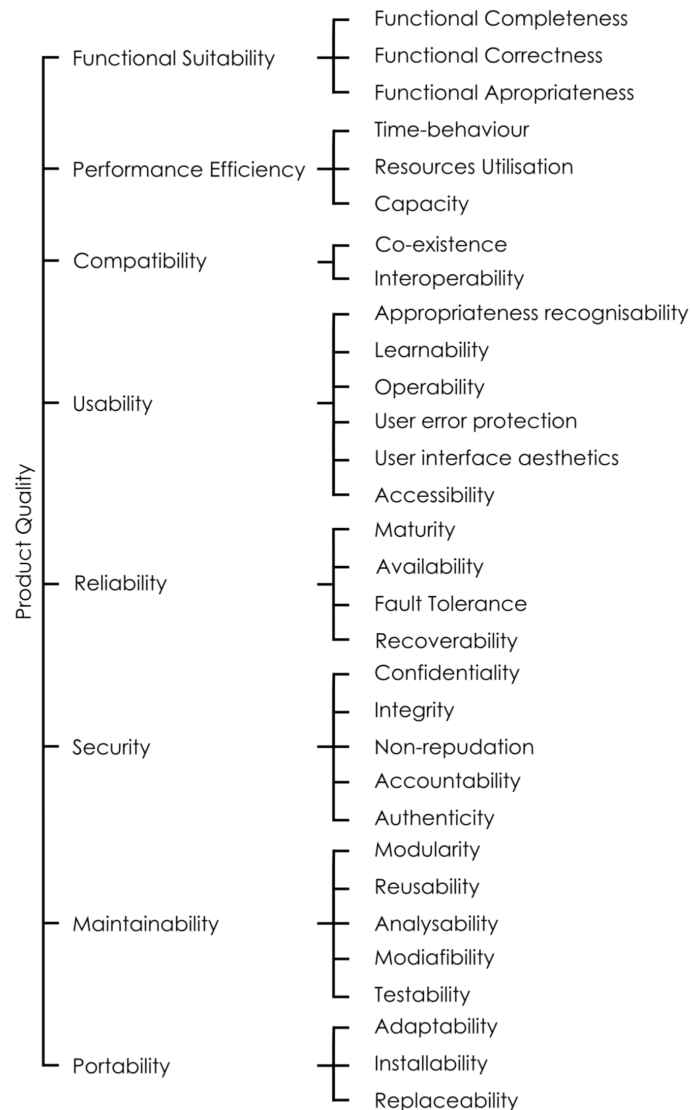

Figure 1.5: SQuaRE's product quality characteristics

ing directly from the ISO/IEC 9126 quality model, have been developed in order to address more specific domains. These "domain-specific quality models" are numerous and intend to address specific elements (products or processes) in a more focused way. Among others, an attempt of quality model designed specifically for model-driven engineering can be found in [Mohagheghi and Dehlen, 2008], a model that addresses software product lines is introduced in [Trendowicz and Punter, 2003] and a model that focuses on the conceptual modelling of data models (ERA) in [Moody and Shanks, 2003].

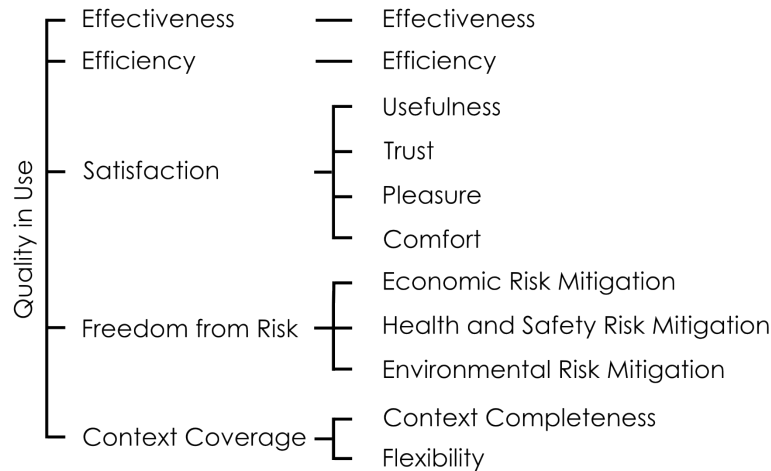

Figure 1.6: SQuaRE's quality in use

1.1.4 Other quality frameworks

Hierarchical quality models are not the only structure used to define software quality. The research works presented below are examples of quality frameworks that diverge from those "divide and conquer" approaches.

Perhaps the most anti-hierarchical take on quality assessment is the use of Bayesian Belief Networks (BBN) introduced in [Neil and Fenton, 1996] and [Neil et al., 2000]. Basically, a BBN is a graphical network and each of its nodes is a probabilistic variable while each of its edges is a causal link between variables. Each node is associated with conditional probability functions that model the uncertainty of the relationships between nodes. According to [Neil and Fenton, 1996], the use of BBN in quality assessment provides several advantages (e.g., association of intuitive graphical representation with underlying mathematical basis, ability to use facts but also expert opinions as 'metrics', possibility to create and use complex models). This approach has been extended in order to decrease the need to develop a specific BBN for each new development [Fenton et al., 2007].

Some quality frameworks also rely on the semiotic theory. These frameworks are used in the evaluation of conceptual models (from database relational schemes to UML-based architectural diagrams). [Lindland et al., 1994] proposes such a framework. It borrows three linguistic concepts (i.e., syntax, semantics and pragmatics) according to which a given conceptual model will be assessed. As shown in Figure 1.7, the syntactic aspects are concerned with the relationship between the conceptual model and its language (e.g., syntactic correctness). The semantic aspects address the relationships between the domain that is modelled

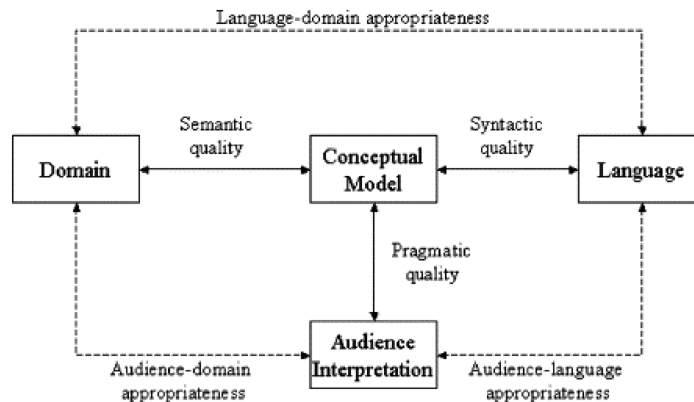

Figure 1.7: Quality in conceptual modelling

and the model itself (e.g, completeness). The pragmatic aspects are focused on the target audience and its interpretation of the model (e.g., comprehension). The framework introduces a clear differentiation between the goals (what quality factors are desired) and the means (how this quality goal will be achieved) and also introduces the notion of feasibly of these goals. [Krogstie et al., 1995] introduces an extended version of this framework which provides additional aspects (physical quality, perceived semantics quality and social quality).

1.2 Software Measurement

1.2.1 Fundamentals of software measurement

The basics of software measurement (i.e., the characterisation of abstract concepts) appeared in social sciences well before the idea of measuring software emerged (e.g., in [Stevens, 1975]). Basically, fundamental measurement can be defined as *a means by which numbers can be assigned according to natural laws to represent the property, and yet which does not presuppose measurement of any other variables than the one being measured*, according to [Torgerson, 1958]. Software measurement is not so much about natural laws than it is about characterisation. The definition provided in [Fenton and Pfleeger, 1998] is therefore more adequate:

> Formally, we define measurement as a mapping from the empirical world to the formal, relational world. Consequently, a measure is the number or symbol assigned to an entity by this mapping in order to characterise an attribute.

This mapping relationship between empirical world and measurement is formalised in [Chirinos et al., 2005], as shown in Figure 1.8.

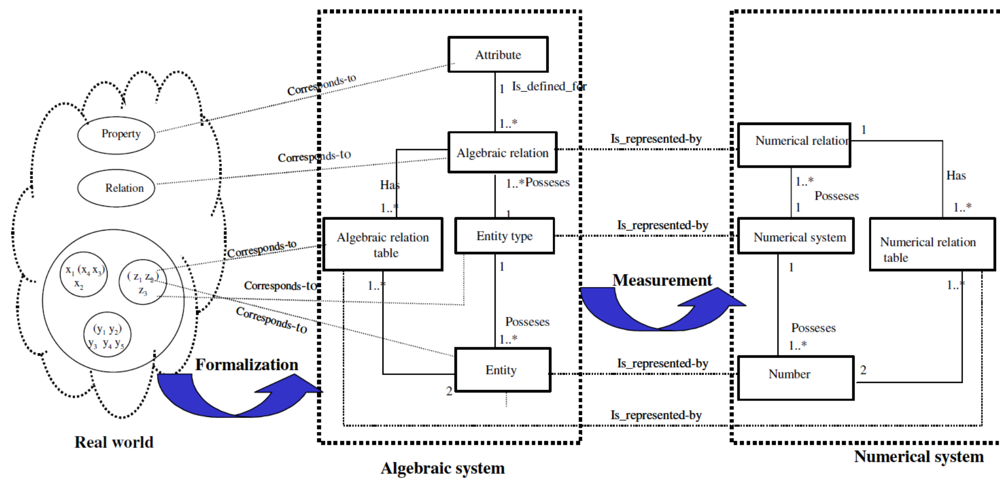

Figure 1.8: Relationship between real and formal worlds through measurement.

Representational theory

[Basili and Weiss, 1984] proposes a first methodology for the collection of software measurement data. From this point on, many measurement definition, validation or exploitation frameworks have been proposed. Their goals are either to structure and/or analyse measurement methods and discuss their validity issues.

[Kitchenham et al., 1995] proposes such a framework. It establishes both theoretical and empirical methods for validating the properties of the elements of the measurement and the models used to define those elements. [Fenton and Pfleeger, 1998] provides a coherent and rigorous framework for controlling, managing, and predicting software development processes. Other notable software measurement frameworks may be found in [Jacquet and Abran, 1997], in [Zuse, 1997] or in [Lopez et al., 2003]. All these frameworks rely on the representational measurement theory. This theory [Morasca, 2001]:

> formalises the 'intuitive' empirical knowledge about an attribute of a set of entities and the 'quantitative' numerical knowledge about the attribute. The intuitive knowledge is captured via the so-called empirical relational system and the quantitative knowledge via the so-called numerical relational system. Both the empirical and the numerical relational systems are built by means of set algebra. A measure links the empirical relational system with the numerical relational system in such a way that no inconsistencies are possible, as formalised by the Representation Condition. In general, many measures may exist that quantify equally well one's intuition about an attribute of an entity (e.g., weight can be measured in kilograms, grams, pounds, ounces, etc.).

Axiomatic approaches

In parallel to the frameworks based on the representational theory, many research efforts (called axiomatic or property-based approaches) have been carried out to develop new approaches to the definition and validation of measurement methods. The axiomatic approaches differentiate themselves from the other frameworks by the way they describe the expected properties of measures. The aim is to describe the characteristics of the measures defined for software attributes via mathematical properties that they should satisfy, while relying on an abstract description of the software artefacts.

Early attempts can be found in [Prather, 1984], in [Weyuker, 1988] and in [Tian and Zelkowitz, 1992] which all focus on single attributes (mainly complexity) or general properties for software measures. An effort to provide a precise mathematical definition of several attributes (size, length, complexity, cohesion, coupling) following this axiomatic approach can be found in [Briand et al., 1996] and a stable and usable validation framework that relies on axiomatic approaches was introduced in [Morasca and Briand, 1997]. This second main branch of measurement definition and validation is still constantly evolving and introduces new refinement, as in [Morasca, 2008].

However these two categories of approaches to the definition of software measurement methods coincide on the fact that a measure (or metric) is defined in order to characterise an attribute of an entity and that some measures can be more or less adapted to a given attribute.

Metrology and unified terminology

Finally, it is worth mentioning efforts, such as [Abran and Sellami, 2002], intending to tie software measurement with metrology (i.e., *"the science of measurement, embracing both experimental and theoretical determinations at any level of uncertainty in any field of science and technology"* [ISO/IEC, 2007b])) concepts or aiming at the unification of the software measurement terminology, such as [García et al., 2006] or [Habra et al., 2008]. Similarly, the ISO/IEC 153939 standard [ISO/IEC, 2007a] recently included in the SQuaRE standard builds on a terminology that is for the most part aligned to the metrology vocabulary [Abran, 2010].

1.2.2 Software Measures

Although the initial concern of Software Measurement was to address the source code, many attempts to provide reliable measures for various software products have been carried out in the past two decades. As a result, the field of Software Measurement regroups many more or less validated measure proposals, aiming to characterise many different types of entities (e.g., code, diagrams, etc.).

Requirements and specifications

Although a widely used measure for requirements is the count of their number as an estimation of size, effort or cost [Morasca, 2001], this method presents several limitations (e.g., the fact that the result and its soundness is very sensitive to the level of granularity of the requirements) and has rapidly been calling for improvements.

Originally introduced in [Albrecht, 1979], function points are designed to extract information from the requirements and have been used as a measure of several attributes [Abran and Robillard, 1994] including, size, productivity, complexity, functionality and overall behaviour, to name a few. Despite some theoretical problems, function points are widely spread and have undergone several variations: Mark II Function Points introduced in [Symons, 1991], COSMIC-FFP which has become a standard for functional size estimation [ISO/IEC, 2003b] and various others whose description can be found in [Bundschuh and Dekkers, 2008], among others.

Besides, a number of metrics have been defined to address UML uses cases. Most notably, [Marchesi, 1998] introduces an indicator of complexity based on the use cases while [Saeki, 2003] provides a set of metrics designed to be indicators of the modifiability of the system. Other sets of metrics have been proposed and a complete study can be found in [Genero et al., 2005a].

In the meantime, software specifications, due to the fact that they are often written in plain text, received few attention regarding measurement methods. Measurement methods have nonetheless been defined for some formal and semi-formal types of specification. Notably, [Briand and Morasca, 1997] presents a preliminary study that was carried out on TRIO+ specifications (a formal object-oriented specification language) where internal attributes are used as quality indicators. [Boloix et al., 1993] defines measures for specifications written with data flow diagrams. An attempt to define a measurement method addressing the *comprehensibility* (attribute) of a specification written in Z language is discussed in [Finney et al., 1998]. Finally, we may also mention a proposal of measures defined for a number of internal attributes (i.e., size, length, complexity, and coupling) of software specifications written with Petri nets in the context of concurrent software systems, which is introduced in [Morasca, 1999].

Conceptual models and high-level design

The attention paid to design activities in the development life-cycle has been increasing steadily since the beginnings of Software Engineering. From a secondary role as documentation, the elaboration of conceptual models has evolved and is now considered as a central and full-fledged activity (due, among others, to the apparition of model-driven engineering). It is thus not surprising that

the amount of proposed measurement methods linked to the conceptual steps of software development has increased consequently.

One well-known metrics proposal can be found in [Chidamber and Kemerer, 1994]. The proposal contains six metrics (Weighted Methods per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling between Object Classes (CBO), Response for a Class (RFC) and Lack of Cohesion of Methods (LCOM1)) designed for both code and high-level design. Only three of them can be applied to UML class diagrams (WMC, DIT, NOC). This suite of metrics mainly addresses the *complexity* (attribute) of design with the purpose of tying it to quality characteristics such as maintainability or reliability. They are applied to (object-oriented) classes. Several empirical validations have been carried out to verify this set of metrics.

[Li and Henry, 1993] introduces another set of metrics defined at the class level for coupling, complexity and size. Although some of those metrics do not satisfy expected properties of the property-based framework described in [Briand et al., 1996], the metrics have been successfully applied to estimate the maintenance effort of real systems. [Lorenz and Kidd, 1994] introduces other metric proposals focusing on static characteristics of software design. These two sets address internal aspects of the classes and not only their external organisation.

It is worth mentioning that [Brito e Abreu and Carapuça, 1994a] introduces a complete collection of candidate metrics for both high-level design and code defined for several attributes (design, size, complexity, reuse productivity and quality) within the scope of a classification framework (TAPROOT). The well-known MOOD suite is the next contribution from these authors. Introduced in [Brito e Abreu and Carapuça, 1994b] and improved in [Brito e Abreu and Melo, 1996], the purpose of this set of metrics is to explore typical object-oriented mechanisms (i.e., inheritance, polymorphism and information hiding) and their impact on quality and development productivity. The suite has been theoretically validated using Kitchenham's framework [Kitchenham et al., 1995] and applied in the context of several empirical studies. A MOOD2 version has been introduced but neither theoretically nor empirically validated, according to [Genero et al., 2005b].

Efforts concentrating on *cohesion* and *coupling* (attributes) of the high-level design of an object-based system can be found in [Briand et al., 1997a], in [Harrison et al., 1998] and in [Briand et al., 1999]. [Bansiya et al., 1999] and [Bansiya and Davis, 2002] address *encapsulation*, *composition* and *inheritance* beside the previous two attributes. We can also mention an attempt to adapt function point analysis to the evaluation of object-oriented design in [Antoniol et al., 1999].

All the above proposals are not specifically designed for class diagrams and, in most cases, they are designed to address both object-oriented design and code. Conversely, [Marchesi, 1998] introduces a first effort to define UML-specific mea-

sures. This proposed set of metrics addresses *complexity*, *cohesion* and *coupling* attributes as well as the *responsibilities balancing*. Little theoretical or empirical validation has been provided regarding this set of metrics.

[Genero et al., 2000] and [Genero, 2002] provide a set of metrics assessing the *complexity* (attribute) of UML class diagrams. These metrics have been theoretically validated through axiomatic approaches and measurement theory. Their use as an early indicator of the maintainability quality characteristic of class diagrams has been discussed and empirically validated in [Genero et al., 2007].

Further details on measurement methods designed for UML class diagrams can be found in [Genero et al., 2005b]. Additionally, an important research work can be found in [Lange, 2007], regarding the assessment and improvement of UML modelling. This work is connected with other efforts related on different levels, such as [Lange and Chaudron, 2004] which explores the *completeness* (attribute) of UML models and the way to assess it or [van Opzeeland et al., 2005] which investigates the correspondence between UML designs and their implementations, to name a few.

Metrics have also been proposed for UML statechart diagrams. In [Miranda et al., 2003], a set of metrics addressing their *complexity* and *size* is defined and, more recently, [Cruz-Lemus et al., 2009] addressed the assessment of the *understandability* of statecharts diagrams. An attempt to formalise statechart diagram metrics using OCL expressions has been introduced in [Reynoso et al., 2008]. It is also worth mentioning the existence of measurement method proposals for statecharts not linked to UML, as in [Derr, 1995].

Finally we can also mention some efforts to assess the quality of database conceptual models. Notably, [Si-Said Cherfi et al., 2007] proposes a set of metrics (addressing *clarity*, *simplicity*, *expressiveness*, *minimality*) applied to different versions of Entity-Relationship conceptual schemas. [Genero et al., 2005a] also addresses this issue.

Low-level design and code

A widespread measurement proposal for low-level design is found in [Henry and Kafura, 1981]. The Information Flow Complexity (IFC) is based on the fan-in (the input parameters and the global data structures from which the function retrieves information) and the fan-out (the output parameters and the global data structures that the function updates) of functions.

At the code level, many measurement methods are found, from the very common yet controversial number of lines of code defined for the *size* (attribute) of code to much more elaborate metrics. Among others, we can mention the following research works.

One well-known measurement method that has been defined to assess the complexity of source code is the Cyclomatic Complexity proposed in [McCabe, 1976]. This measure is based on the control flow graph of the program. It relies on the assumption that the higher the number of paths in a program is, the higher its control flow *complexity* will be. The computation of the cyclomatic number is based on graph theory results.

[Halstead, 1977] proposes a set of metrics designed for several attributes of the source code (e.g, program length, length estimator, volume, potential volume, an more). However, according to [Morasca, 2001]:

> Halstead's Software Science's theoretical foundations and derivations of measures are somewhat shaky, and it is fair to say that not all of the above measures have been widely used in practice. However, due to their popularity and the availability of automated tools for computing them, some of the above measures are being used.

Object-oriented programming also constitutes the right field for a vast amount of software metrics. As a matter of fact, most of the above measurement methods addressing high-level object-oriented design can be used at the concrete code level (and were generally defined for C++ language).

Other measures

It is worth mentioning the large number of metrics defined in the ISO/IEC 9126 standard, dispatched in [ISO/IEC, 2001b], in [ISO/IEC, 2001c] and in [ISO/IEC, 2001d]. These metrics assess all parts of the software products, from design to code and runtime behaviours and are tied to the quality characteristics defined in the ISO/IEC 9126 quality model.

Besides, many other types of software-related entities could be (or even have been) assessed from the documentation-related products (as in [Matulevicius et al., 2009]) or test-related products (examples may be found in [Morasca, 2001]). Similarly, a proposal of metrics designed to relate OCL expressions to cognitive complexity is introduced in [Genero et al., 2005a]. Another example is the proposal of metrics designed to address the model transformations that can be found in [van Amstel et al., 2009].

1.2.3 Implementation of measurement programs

In spite of the wealth of quantitative approaches and generic frameworks available, the development and successful use of measurement in actual situations still remains a difficult and demanding task. The research works presented in this section focus on the operationalisation of software measurement and quality assessment. Some of them focus on the generation of customised measurement

plans (i.e., measurement plans that are specifically adapted to a given environment), others address the theoretical or logistical support essential to a successful quality assessment program.

GQM and GQM/MEDEA

The first fundamental rule to deploy a successful measurement plan is that the purpose of any measurement should be clearly stated from the beginning. Described in [Basili et al., 1994], the Goal/Question/Metric (GQM) approach relies on an application of this rule. It provides a framework that helps define the relevant measures from predefined measurement goals. The definition of a measurement goal is provided according to five dimensions (Object of Study, Purpose, Quality Focus, Point of View, Environment). This goal is then translated into relevant questions (e.g., how high is the defect density?) that are refined in metrics destined to provide the answer to this question. This approach has several advantages: it is environment specific (i.e., the goals and questions are designed for a given context), it integrates easily into the development process and reverses the usual bottom-up software measurement approaches (i.e., everything that can be measured is measured and the conclusions are drawn afterwards). However, it lacks the support to ease the derivation of metrics from the initial questions (which is not trivial) and is not related to any specific quality framework or model. Several refinements of GQM have been proposed in order to address the difficulty to define the measures according to goals, such as an attempt of automated support in [Lavazza and Barresi, 2005] or the Goal/Argument/Metric(GAM) described in [Cyra and Górski, 2008].

Among these refinements of the GQM approach, the GQM/MEDEA framework builds on the goal-driven definition of measures, coupled with a set of empirical hypotheses. The aim of the approach is to provide a *"measure definition process, usable as a practical guideline to design and reuse technically sound and useful measures"* [Briand et al., 2002]. The framework provides a detailed description and an information flow of the various activities involved in the definition of measures. Measures are links to corporate goals and the development environment, and the overall approach helps justify, interpret, and reuse measures, as well as *"identify problems that may arise during the definition of measures, taking into consideration that it is a highly human-intensive process"* [Briand et al., 2002]. The other notable feature of the framework is the introduction of a conceptual model (shown in Figure 1.9) that constitutes the foundation of a repository designed to contain all the knowledge relevant to measurement and therefore poses the bases of the metamodel-based approaches described in Section 1.4.

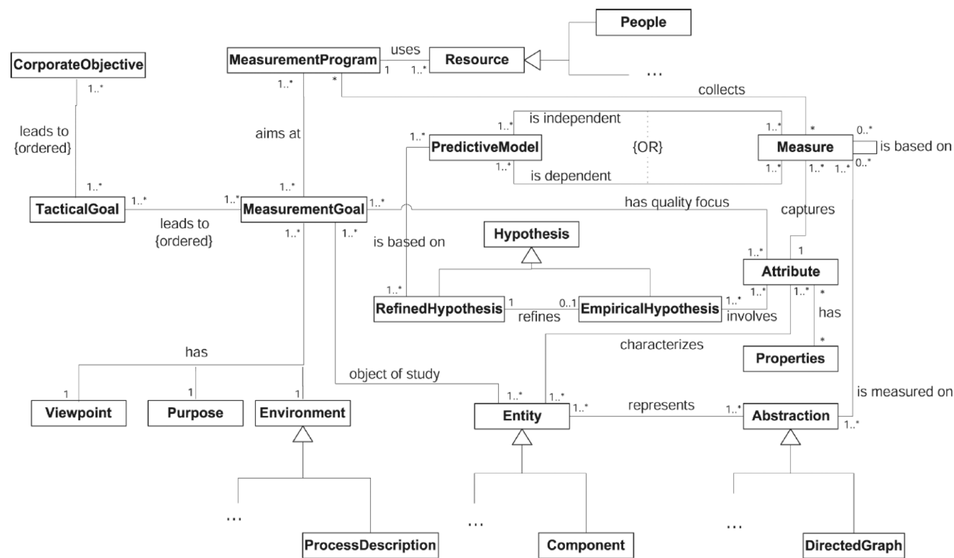

Figure 1.9: GQM/MEDEA conceptual model

SQUID and MOSME

The SQUID (Software QUality In the Development process) approach, defined in [Bøegh et al., 1999], intends to support the quality management during the development process through defined activities and a provided tool-set. It combines the process and product approaches to software quality and is based on three models (a product view, a data view, and a quality view of software) that are connected by means of software measurement. Three entity types (deliverables, activities, events) are considered in the product view and provide a description of the software development process. The quality view defines software quality characteristics, sub-characteristics and attributes (measurable properties). This view is connected with the ISO/IEC quality model and its own definition of software product. The data view is in charge of the data elements to be collected and divides them into three categories : actual, target and estimate values.

A notable research work inspired by the SQUID approach can be found in [Chirinos et al., 2005]. This approach also proposes a data model for software measurement (shown in Figure 1.10). This Model for Software Measurement (MOSME) intends to define explicitly software measures, providing a more structured view than (yet compatible with) SQUID. It intends to address the problem *"of constructing software measures to obtain reliable, repeatable and comparable values"*. The MOSME data model focuses on the definition and modelling of the elements involved in software measurement, particularly the counting rules and the role played by the context of use.

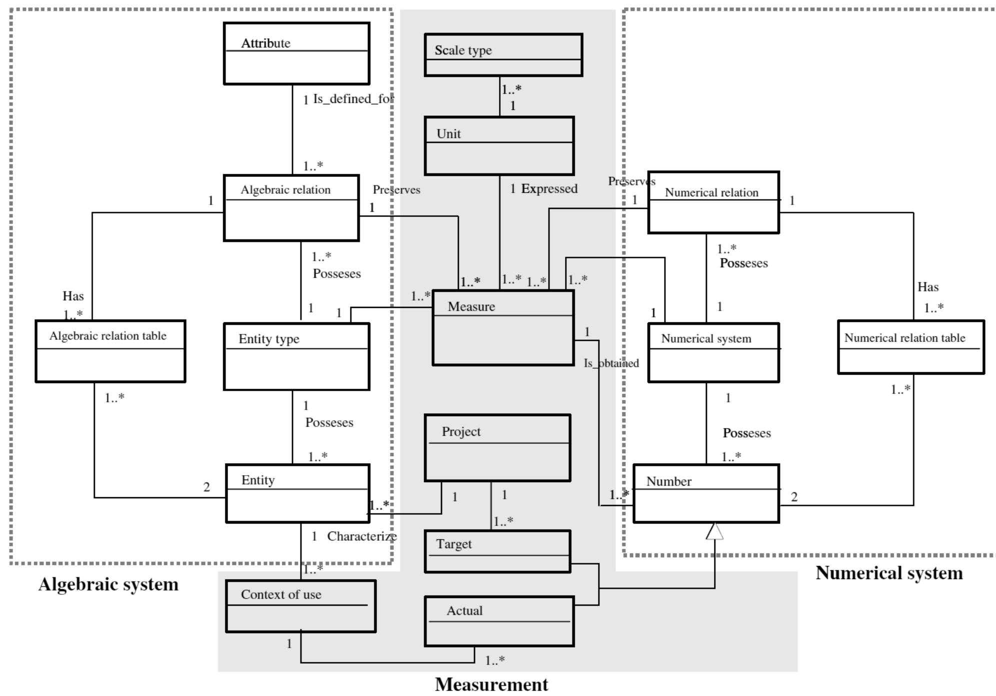

Figure 1.10: MOSME data model

ISO/IEC 15939 and MIM

The ISO/IEC 15939 standard *"identifies the activities and tasks needed to successfully identify, define, select, apply,and improve software measurement within an overall project or organisational measurement structure"* [García et al., 2006]. The standard relies on two components to structure the measurement definition and exploitation activities: a software measurement process that defines the activities involved in the measurement process and a conceptual model that structures the various elements involved in this process. Additionally, it provides a terminology of measurement-related terms commonly used in the software industry and mostly aligned with the concepts of metrology [Abran, 2010].

The software measurement process defined in ISO/IEC 15939 relies on the design of an *information product*, that is, a set of measures and indicators defined to satisfy an *information need* expressed by an individual or group of individuals involved in the software development. It emphasises the difference between purely measurement-related activities (i.e., the data collection and data preparation) and the definition and interpretation of indicators in order to satisfy information needs (i.e., quality assessment). As shown in Figure 1.11, the Measurement Information Model (MIM) defines and structures the relationships between measures and information needs. It also formalises how attributes are combined in order to provide an indicator that satisfies a specific information need.

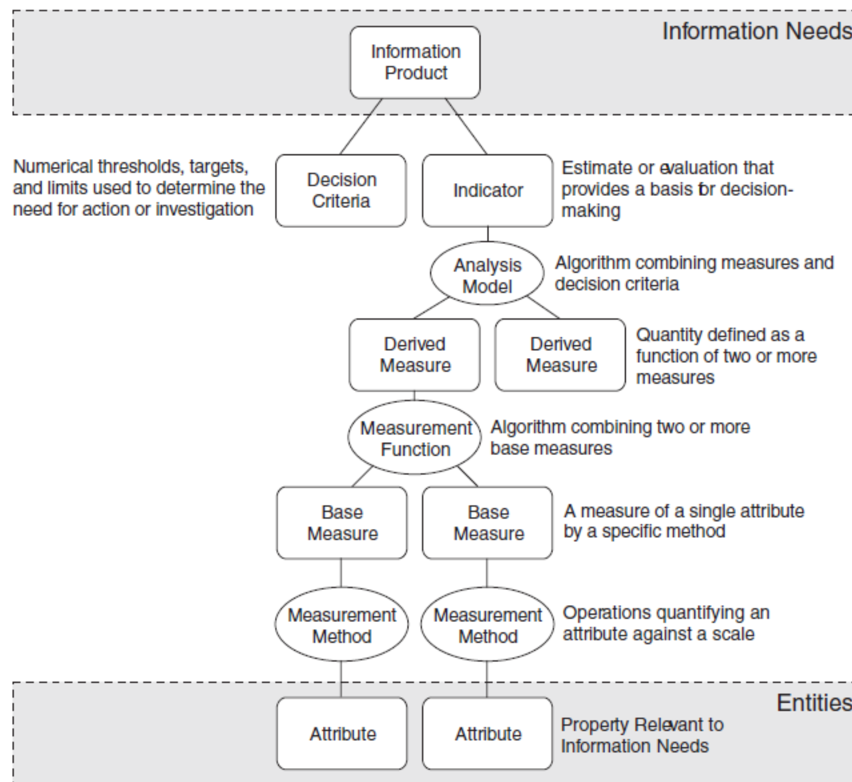

Figure 1.11: MIM conceptual model

1.3 Software process improvement

Process improvement consists in the definition of a series of actions taken in order to analyse and improve existing business processes so that the organisation meets its goals (e.g., increasing profits and performance, reducing costs, etc.). Although the focus of software improvement is management-oriented, it is also regarded as *"a method to introduce process changes to improve the quality of a product or service, to better match customer and consumer needs"* [Cook, 1996]. As such, several software process improvement frameworks have emerged or adapted, aiming to improve the quality of software processes and, therefore, products.

Total Quality Management

Total Quality Management (TQM) relies on the continuous application of quantitative methods and the use of human resources to improve the material and services supplied to an organisation, in turn increasing the level of satisfaction of the customers at a steady rate, until fulfilled [Li et al., 2000].

TQM also adopts a specific view on quality in which the customer is the final arbiter. The methodology may be regarded as customer-driven. It emphasises

the continuous process improvement to achieve high quality products or services, relying on the assumption that a better process will contribute to the improved "total quality" of the organisation and, therefore, the quality of the final product.

[Deming, 2000] proposes 14 points of actions allowing to implement TQM. These 14 points may be adapted to software development, as explained in [Li et al., 2000]. Most of them consist in the definition of a suitable work environment (e.g., "drive out fear of job insecurity", "eliminate slogans, exhortations, and targets for the workforce"). More importantly, one of these points proposes to eliminate "quotas" (i.e., schedule and metrics), stating that the metrics are counter-productive if used as a control method.

TQM, which is arguably more of a philosophy or of the adoption of certain work ethic, has helped many companies to improve quality of products and processes, and in turn, increase the productivity and the profitability [Li et al., 2000]. However, the methodology is not void of flaws. Regarding the drawbacks of TQM, [Li et al., 2000] states:

> One caveat is that there is no free lunch for those who perform TQM activities. Once you implemented TQM concept and methods, you are bound to continually improve your products and processes. You must constantly ask yourself "What and how can I do it better next time?" [...] Most importantly, there is no such thing as few (i.e., the management) or mindless majority (i.e., the workers). Everyone related to the value chain of the product is significant and must use his or her mind constantly to play his or her own role well, otherwise, the chain will be broken, and the TQM process will soon fall apart.

Therefore, implementing TQM is a process that must be carried out with a small number of people, keeping the process manageable. Similarly, this process improvement methodology requires a strong emphasis from the top management in order to keep all employees motivated and willing to adhere to the underlying work ethic.

Six Sigma

Directly inspired by TQM, the Six Sigma management strategy seeks to identify and eliminate causes of errors/defects/failures in business processes. Six Sigma relies on a focus on customers, a process orientation and a leadership based on metrics. It aims to eliminate defects (i.e., anything which could lead to customer dissatisfaction according to the approach) using the application of statistical methods. The fundamental objective of the Six Sigma methodology is the implementation of a measurement-based strategy that focuses on process improvement and variation reduction [Antony, 2004].

This emphasis on metrics coupled with the focus on customers is particularly relevant in the context of software. As explained in [Biehl, 2004]:

> By building critical customer metrics into software solutions (for example, response times, cycle times, transaction rates, access frequencies, and user-defined thresholds), [software engineers] can make applications self-correcting by enabling specific actions when process defects surface in the improvement zone. These actions do not always need sophisticated technical solutions to be beneficial.

Among the main limitations of Six Sigma, the difficulty to obtain quality-related data remains an hindrance that needs to be overcome since many processes do not provide quantitative data, although they are effort and time-consuming [Antony, 2004].

ISO/IEC 9000 standards

ISO/IEC 9000 standards provide a series of rules designed to formally organise processes to manufacture products while managing and monitoring progress. These rules (called requirements) help ensure that the output (i.e., products or services) of the organisational process meets the expectation of the customers, that the quality system is consistently implemented and verifiable, that measures are collected to demonstrate the effectiveness of various aspects of the system and that the continuous improvement of the company's ability to meet customer needs is respected [Kantner, 2000].

Although the standards were originally created for the manufacturing sector, ISO/IEC 9000 standards have been applied to software development as well. As such, the implementation of ISO/IEC 9000 relies on the ISO/IEC 9126 standard described in Section 1.1. Although the process of implementing the ISO/IEC 9000 standards is a rigorous process that may increase the costs, the added value or ISO/IEC 9000 lies in the impact is has on the organisational culture. Indeed, according to a survey reported in [Stelzer et al., 1996]:

> It seems that it is not the technical contents of the IS0 9000 family that makes it specifically appropriate for software process improvement. The culture created by a company-wide improvement program seems to be more important.

SPICE

Defined specifically for software processes, the ISO/IEC 15504 standard, also known as Software Process Improvement Capability Determination (SPICE), is *"the reference model for the maturity models (consisting of capability levels which*

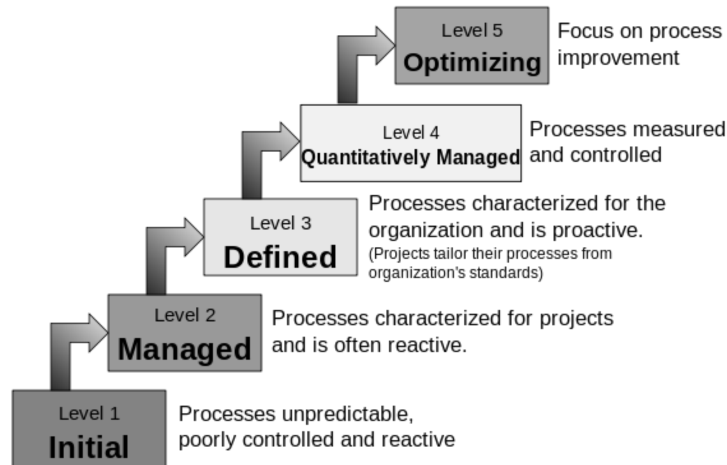

Figure 1.12: CMMI maturity levels

in turn consist of the process attributes and further consist of generic practices) against which the assessors can place the evidence that they collect during their assessment, so that the assessors can give an overall determination of the organisation's capabilities for delivering products (software, systems, and IT services)" [ISO/IEC, 2003a].

As explained in [Paulk, 1999], SPICE intends to help characterise the *process capability* through a series of nine process attributes, applicable to any process. These attributes represent measurable characteristics that help manage a process and improve its capability to perform. Each process attribute describes an aspect of the overall capability of managing and improving the effectiveness of a process, in achieving its purpose and contributing to the business goals of the organisation. The process attributes are grouped into capability levels (ranked from 1 to 5). Capability levels *"constitute a rational way of progressing through improvement of the capability of any process* [Paulk, 1999].

CMMI

The Capability Maturity Model Integration (CMMI) is a framework that describes the principles and practices designed to lead to software process maturity. It is intended to *"help software organisations improve the maturity of their software processes in terms of an evolutionary path from ad hoc, chaotic processes to mature, disciplined software processes"* [Paulk, 1999].

The underlying principles of CMMI are similar to those of SPICE. CMMI is organised into five maturity levels, as shown in Figure 1.12. Each maturity level is decomposed into several key process areas that indicate the areas an organisation should focus on to improve its software process. CMMI also relies on measurement

in order to monitor the maturity level. Detailed measures of software processes and products quality are required at level 4. In order to satisfy to this level of maturity, both the software process and products have to be quantitatively understood and controlled.

According to [Staples et al., 2007], the main reason for organisations not to adopt CMMI is their (small) size, the costs induced by the process, the time needed to implement the framework or the existence of other SPI methods in their context.

1.4 Quality modelling

We classify under the quality modelling category all efforts consisting in explicitly modelling an aspect of the quality assessment process in order to guide this process. The concept of quality modelling originates in fact from GQM [Basili et al., 1994] described in Section 1.2. Although it is mainly an approach to the implementation of measurement plans, GQM proposes to explicitly derive any planned measurement from a hierarchy of quality goals expressed through more concrete questions, therefore defining a hierarchical model of the quality assessment process. During the past few years, several research efforts regarding quality metamodels or software measurement modelling have been carried out and contributed to advance the topic of quality assessment modelling.

1.4.1 GenMETRIC and SMML

Relying on a previous effort to describe a consistent terminology for software measurement (in [García et al., 2006]), [García et al., 2007] provides an approach to measure definition supported by metamodel. The approach provides a metamodel (shown in Figure 1.13) that captures all the relevant concepts of measurement theory and hierarchical quality models in order to describe software measurement models for an entity type. The description of the entities and the measures that can be applied to them is provided by a distinct metamodel (e.g., relation database metamodel for a database schema). The framework is supported by a tool (GenMETRIC) that allows the generation of software measurement models and the calculation of measures.

The main advantage of this approach is to provide a generic and flexible environment for software measurement which is not restricted to only one kind of products or to a single quality model. The approach has already been adapted successfully to various domains [Cachero et al., 2007].

Building on this metamodel, [Mora et al., 2008] intends to provide a simple and intuitive procedure to design a measurement model (i.e., a model that defines all the relevant elements of a measurement plan). To this end, it introduces a graphical domain-specific language (SMML) that is dedicated to this

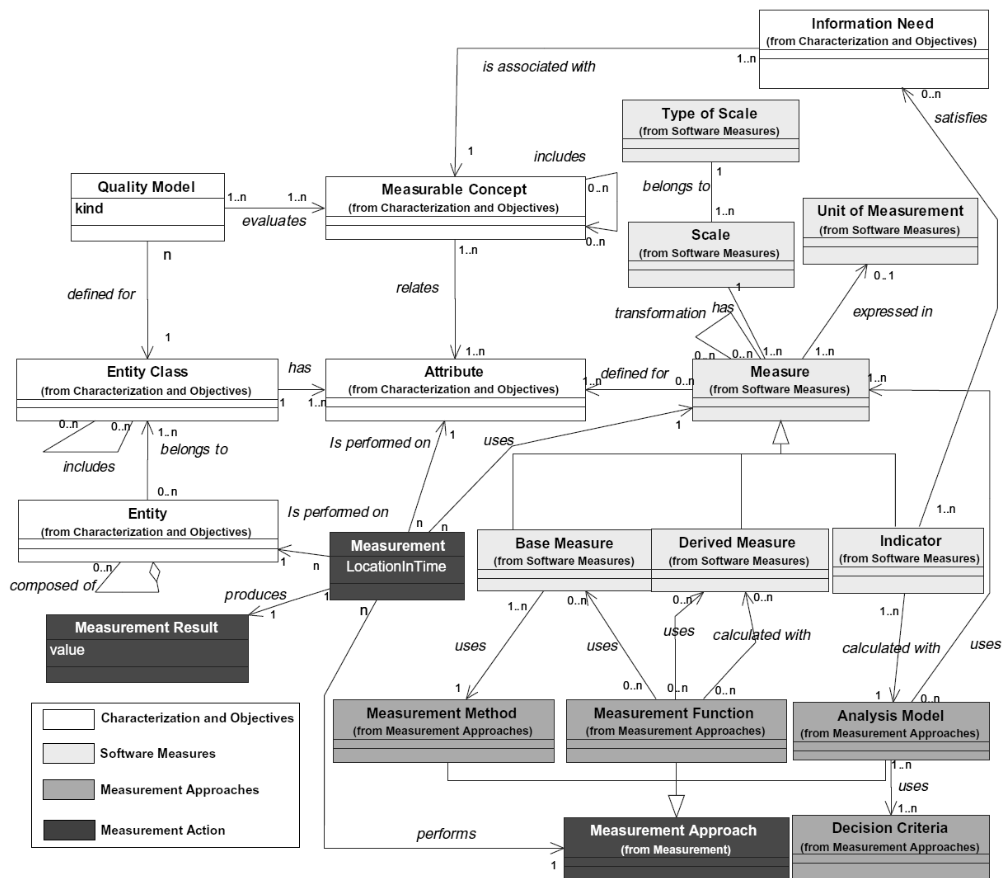

Figure 1.13: GenMETRIC underlying metamodel

task. SMML provides a set of pictorial representations of measurement-related concepts and allow the graphically combination of them in order to provide a model representation of what to measure and how to measure it.

1.4.2 QMM and Quamoco

[Deissenböck, 2009] provides an approach to the maintenance of software that also relies on a quality metamodel (QMM). This metamodel is not inspired directly by any of the proposals presented in Section 1.1 and do not focus heavily on the concept of software measurement. However, a vast effort aiming at the operationalisation of the quality model is provided as well as the description of a tool that supports the overall process.

Extending the scope of QMM, [Wagner et al., 2012] introduces a more generic quality metamodel, addressing any quality factor. It also proposes an approach (Quamoco) design to support the continuous assessment of Java and C# systems. The expressiveness of the quality metamodel (i.e., the ability of the quality

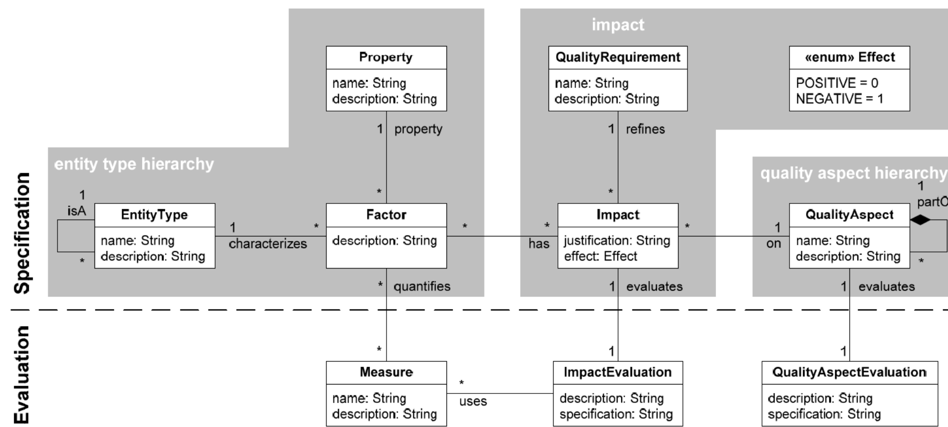

Figure 1.14: Quamoco metamodel for specifying and evaluating software quality

metamodel to express several quality models) has been evaluated in [Klaes et al., 2010].

1.4.3 Other quality metamodels

The support provided by a metamodel is also a common aspect of the following efforts. In [Mens and Lanza, 2002], a metamodel supports the precise definition of the artefact to measure (object-oriented systems to be more specific). This metamodel and the graph-based description of generic metrics allow the generation of typical object-oriented metrics.

[Lee and Chang, 2000] introduces RAMOOS, a tool support for the utilisation of customised quality models in an object-oriented context, and [Khosravi and Guéhéneuc, 2005] also stresses the importance of tool-supported customised quality models, while not relying on a metamodel to generate these quality models.

[Dubielewicz et al., 2006] provides a quality metamodel for requirements evaluation and assessment that is mainly inspired by the ISO/IEC quality model. This metamodel allows the generation of quality models that are adapted to various situation.

[Mohagheghi et al., 2008] builds on the quality framework introduced in [Mohagheghi and Dehlen, 2008] and defines a quality metamodel to support the framework. This metamodel does not align on the usual terminology of software measurement.

Finally, a quality metamodel addressing design rationale (i.e., the decision taking in the course of a model-driven engineering process) is introduced in [García Frey et al., 2011]

Chapter 2

Research issues

Chapter 1 showed that numerous efforts have been carried out in order to advance the field of Software Quality. However, many limitations still remain and prevent software quality assessment to leverage its full potential as a core activity of Software Engineering. This chapter explores the main issues researchers are still striving to address. Section 2.1 and 2.2 addresses intrinsic limitations of declarative (i.e., quality models) and analytic (i.e., software measurement) approaches to software quality, respectively. Section 2.3 is dedicated to issues regarding the integration of quality assessment into the software development life-cycle.

2.1 Issues related to quality models

As we have seen, quality frameworks/models have been proposed very early in the evolution process of Software Engineering. However, although they have become part of the software quality landscape, their efficiency is still debated. As explained in [Deissenboeck et al., 2009]:

> Software quality models are a well-accepted means to support quality management of software systems. Over the last 30 years, a multitude of quality models have been proposed and applied with varying degrees of success. Despite successes and standardisation efforts, quality models are still being criticised, as their application in practice exhibits various problems. To some extent, this criticism is caused by an unclear definition of what quality models are and which purposes they serve. Beyond this, there is a lack of explicitly stated requirements for quality models with respect to their intended mode of application.

This observation may in fact be refined into two main issues regarding quality models. The first main issue is related to operationalisation concerns. The other deals with the question of what quality models actually model.

2.1.1 Complexity of the operationalisation

As explained in [Wagner et al., 2009], one of the main limitations of quality models is the difficulty to apply them in an actual software development context and *"make them work in a realistic environment and producing quantified results"*.

As we may see in Chapter 1, hierarchical quality models may be divided into two distinct families. One includes general quality models focusing on the 'software product' (e.g., ISO/IEC-9126). The other includes quality models designed for specific domains or intermediary products (i.e., requirements, design, etc.). However, none of those two perspectives fully address the problem of how to make quality models operational.

General quality models do not provide a sufficient level of detail to allow an easy and quick operationalisation. Although they provide satisfying general references or basis for quality assessment, their scope does not allow for very specific or customised guidelines that would be applicable as-is to any quality-related challenge a development team could encounter.

The specialised quality models are more operational by design since they address smaller topics (e.g., requirements, documentation, etc.) but they induce a multiplication of external models throughout the development life-cycle. This multiplication of specialised models could result in a more difficult management of the quality assessment and a waste of time, particularly if not all the quality factors of the models are priorities in the given context.

These concerns regarding the operationalisation are not only related to the structure of quality criteria they defined but also to the actual measures proposed to assess their satisfaction. On the one hand, some quality models do not provide any quantitative method at all. On the other hand, general quality models such as the ISO/IEC quality model have to comply to their generic scope and cannot provide very specific measures. Finally, some quality models have been defined at an early stage of evolution of Software Engineering and are not always adapted to more recent paradigms (e.g., model-driven engineering, object-oriented programming, etc.).

Additionally, quality models do rarely address the specifics of the environment. As explained in [Dromey, 1996], quality assessment is highly sensitive to context. An efficient quality model should therefore be tailored to take parameters of the actual development environment (priorities, availability of given measurable entities, constraints that influence the interpretation of results, etc.) into account. This limitation is directly observable in the case of the ISO/IEC quality and the important number of customised quality models that have emerged

from the standard, as well as the fact that the customisation process is neither straightforward nor trivial (as explained in Chapter 1).

In consequence, it is arguable that existing hierarchical quality models are at best adapted to one paradigm or product. However, they cannot take the environmental features of a development process (e.g., priorities, development methodology, etc.) into account, nor can they evolve during the development as the priorities or quality information needs change.

This lack of flexibility is reflected by the poor rate of adoption of quality models in very small to small companies. As shown in a survey conducted on 44 small Belgian companies [Perez Garcia et al., 2012], only 19 percent of the respondents stated their reliance on quality models. These results somewhat corroborate the fact that although they provide a general reference to guide quality assessment efforts, quality models do not provide a practical mechanism that may be used directly in a given context, especially if this context does not possess important resources to allocate on a customisation or operationalisation process.

2.1.2 Confusion between quality models and quality modelling

The idea of using models during software development as a tool to circumvent the inherent complexity of a specific process is not new. However, for a long time, models have mainly been used in very specific tasks or to design a particularly complex piece of code [Pressman, 2000]. The emergence of model-driven engineering provided a different approach, putting models in the front of the process as first class entities. Model-driven engineering is a vast field that encompasses various initiatives (one of the best known being OMG's Model-Driven Architecture) addressing many different opportunities to fully exploit models (i.e., automatic code generation, model transformations, etc. [Schmidt, 2006]). Model-driven approaches contributed to a new vision of what software is by focusing on the different models used to elaborate it all along the development process.

As a matter of fact, quality models are more reminiscent of the former paradigm than they are fitting in the model-driven paradigm. They provide a structured set of criteria that represent an attempt to define the concept of software quality, and (possibly) the quantitative methods to evaluate the extent to which the product complies to this predefined characterisation. As such, they provide an ideal vision of quality that one product should reach.

Another aspect of quality models that prevent the use of powerful modelling techniques is their fixity. As explained in [Deissenboeck et al., 2009], *"Although most quality models conform to an implicitly defined metamodel they usually lack an explicitly specified metamodel that precisely defines the set of legal model instances"*. The lack of explicit quality metamodels in hierarchical quality models is coherent with the observation regarding the role of these models. Since they

provide a structured characterisation of the concept of quality, they are logically "set in stone" and not prone to alteration.

In consequence, quality models should not be confused with quality modelling, which would be the modelling of the actual quality-related aspect of a given software product. An actual model of the quality-related aspect of a given software development context would arguably be more helpful to the developers. Such models would have to be specifically generated to adapt to a specific context and to be evolutive in order to create an accurate quality picture of a software project at any point of its life-cycle.

The investigation of metamodelling techniques applied to the field of Software Quality is therefore a promising opportunity that should be pushed further. However, it is crucial not to limit the notion of quality metamodels to the mere recreation of existing quality models. Instead, efforts should be carried out in order to see how quality-related metamodels may help bridge the gap between declarative approach and software measurement in a more flexible way.

2.2 Issues related to software measurement

As shown in Chapter 1, software measurement relies on solid theoretical and mathematical foundations. However, as the need for new types of measure increases, the spread of mistakes or misconceptions in the definition of measures, the lack of validation or the difficulty to implement a satisfying software measurement program still hinder the maturity of this field.

2.2.1 Conceptual misconception pertaining to measurement

One of the main threats to software measurement is the widespread lack of conceptual clarity of metrics [Habra et al., 2008]. Software measurement methods in general lack clarity about the entity they are characterising or the attribute they are supposed to evaluate. This often results in a misuse or misinterpretation of the values produced by measurement methods. This lack of clarity can therefore lead to the irrelevance of any quality assessment made on the bases of these numbers.

Besides, the evolution of Software Engineering induces a lot of possible confusion regarding measurement. While a defined measure may be perfectly fit to assess an artefact complying to a given paradigm, it may not be applicable in others. For instance, the use of McCabe's cyclomatic number may be adapted to procedural programming, but will not provide satisfying results when applied to object-oriented programming languages [Habra and Lopez, 2004].

A precise definition of measures (as well as a common understanding of its conceptual elements among the various stakeholders) and their intent is therefore crucial in order to avoid misuses of software measurement.

2.2.2 Lack of empirical validation

It is easy to confuse mere quantification with actual measurement [Abran and Sellami, 2002]. Whereas numbers may be associated to software products in a number of ways (e.g., score cards, expert's rating, etc.), defining an actual *measure* requires a lot of attention to the rules of metrology. This critical process logically calls for cautious validation, both theoretical and empirical. The question of software measures verification and validation methods has thus been largely investigated in the literature (e.g., in [Kitchenham et al., 1995], in [Fenton and Pfleeger, 1998], in [Zuse, 1997], in [Morasca and Briand, 1997], in [Habra et al., 2008], etc.).

Despite the availability of validation frameworks, most analytical methods still lack comprehensive and structured empirical (or theoretical) validations [Koziolek, 2011]. Validation is a complicated and long process that requires a lot of resources (especially empirical validation). The lack of validation is therefore understandable but contributes to the spread of poor software measures or misused measurement procedures. Besides, as a quickly evolving field, Software Engineering makes it even more difficult to provide a validated and common set of measures since programming or designing paradigms emerge or are slightly altered almost constantly.

As a result, although the systematic theoretical and empirical validations of software measures should continue to be encouraged and carried out, practitioners have to be provided with methods that help deal with the risk of inadequate measures.

2.2.3 Complexity of measurement programs implementation

Even when provided with validated and reliable software measures, actually applying them in a consistent and manageable program remains as challenging. For instance, some elaborated measurement procedure may be difficult to apply, or even inapplicable at early stages of the development.

Similarly, although a goal-driven top-down measurement plan definition provides a clear understanding of the objectives pursued by the measurement process, if the plan is not carefully taken care of, it may results in dead ends at some point (e.g., a goal may rely on a resource that is not available in the context or not measurable in the case of close-source libraries).

Besides, measurement-based approaches are not integrated enough with other types of quality assessment methods, such as scenario-based approaches to architecture quality [Koziolek, 2011]. This lack of integration induces additional effort in order to take advantage of different techniques with their respective pros and cons.

Finally, and as a result of aforementioned shortcomings, software metrics are widely misused or at least underused. In fact, studies (namely [Kasunic, 2006], conducted by the Software Engineering Institute) show that metrics appeal more to management than they do to analysts or programmers. This fact tends to show that software measurement is mainly used as a control means while it would be better used as a guide.

2.3 Issues related to the integration of quality assessment into the software development

Software development life-cycles are already demanding activities on their own. The focus put on quality assessment and software measurement may therefore conflict in many ways. This section addresses the limitations that prevent a facilitated integration of quality assessment as a continued and companion process of the development itself.

2.3.1 Spread of measurement methods

As we have seen in Chapter 1, quantitative approaches are currently witnessing many efforts to produce new and more accurate measurement methods adapted to every type of software artefact. As a result, many options are offered to the quality assurance teams when considering how to evaluate their products. The first caveat in this context is to avoid the confusion about which measures are selected and why. The stakeholders' transversal understanding of the choice and purpose of measures is essential to the successful integration of quality assessment into the development process.

Besides, the lack of visibility of these numerous proposed metrics and the lack of tool support for measure users to decide what metric is more suited and efficient for a specific need is a threat to the efficient quality assessment of software. A more systematic and structured classification of existing measures would be beneficial to the sound use of quantitative approaches.

2.3.2 Problematic role of quality assessment

As explained before, the software development life-cycle is a complex set of activities. During the course of these activities, many quality concerns and requirements may arise. This multitude of quality concerns/requirements throughout various software processes and activities confers a key role to quality assessment. However, and although quality assessment is the one key aspect that cannot be disregard, it is still considered as a parallel or secondary activity. According to [ISO/IEC, 2008], software quality assessment is not an end by itself but a

means to support other software engineering processes and is classified accordingly (i.e., support process and not primary process). As a supporting activity, quality assessment is thus closely linked to the activities it sustains. First, the supported activities influence the way quality assessment is performed. Besides, the results of the assessment impact the way the supported activity is performed. Therefore, quality assessment should have its own life-cycle, allowing revisions and corrections regarding how it is performed.

2.3.3 Impact of model-driven engineering

Model-driven approaches contributed to a new vision of what software is. Unfortunately this change implies a renewed envisioning of measurement as well. Although metrics dedicated to conceptual models have already been proposed (see Chapter 1), the relationship between these and previously proposed measures must be investigated (e.g., which metrics are applicable to which model(s) and under which conditions?). Similarly model-driven engineering has to cope with a view of software that is less "black-box" whereas classical quality models still consider software to be a monolithic "product". Model-driven approaches create the need to envision new ways to conduct quality assessment. In fact, since Model-Driven Engineering copes with different models and handles them as distinguished products, the quality models have to take this into account and to adapt consequently.

2.3.4 Impact of software ecosystems

Extending even more the scope of what software is regarded as, the emerging notion of software ecosystems also impacts the way quality assessment should be performed. One limitation of traditional quality assessment methods (except for software process improvement frameworks) is their product-centric approach. However, the notion of software ecosystem, defined as *a collection of software artefacts and/or projects, developed and co-evolving in the same environment* in [Lungu, 2009], has recently drawn increasing attention. As explained in [Lungu, 2009], "software projects exist in larger contexts ". The analysis of ecosystems requires different kinds of artefacts used and produced during the software development process, beyond source code [Robles et al., 2006]. Quality assessment should therefore not only rely on final products but on all possible artefacts (e.g., source code, design, mailing list archives, etc.) and the various processes (i.e., development and business models) that relate artefacts to each other.

2.3.5 Organisational issues regarding quality assessment

Any software development project takes place in a larger context that motivates its existence. This applies to any context, from the community-based open-source software to the more conventional software development hosted within a company. This larger context inevitability induces some constraints that can hinder sound quality assessment.

First of all, due to the conceptual complexity behind measurement, the communication between stakeholders with different points of view may be difficult. Without a clear understanding of the goals and possible interpretation of measurement values, different stakeholders may not be able to gain a common understanding of the current status of the project and the actions that must be carried out. In consequence, the rationale of the quality assessment process should be clearly stated and easily available for every stakeholders to consult so that they may understand their part in the process.

Another crucial element is the psychological impact of measurement in an organisation. As explained in [Westfall and Road, 2005], people who actually develop and dedicate time to a project may regard quality assessment as a way to control them and judge their abilities. Therefore, quality assessment is often regarded as an inconvenience instead of a supporting process, which partially explains the observation from [Kasunic, 2006], regarding the appeal of software measurement. In consequence, it is crucial to provide methodologies that help present quality assessment as a supporting activity which is beneficial as a guidance mechanism.

2.3.6 Cost and effort of quality assessment/improvement

Finally, a recurrent problem of quality assessment is the time and effort it requires and, ultimately, the cost it induces. As explained in [Fenton and Neil, 1999], an enduring observation regarding measurement plans remains the overall misguided and/or inefficient use of measures in industry, leading to costly or useless measurement plans.

This waste of time, effort and money is mainly due to the lack of flexibility of existing quality assessment methodologies. Quality models may require lots of customisation and operationalisation in order to fit a context, or induce the use of multiple specialised models.

Measurement values are time-consuming when it comes to their collection and, if not defined adequately, may result in unexploitable output. The more established software measures also require a level of maturity from the project that postpone meaningful assessment until late in the development process, which may result in dramatic efforts of maintenance and corrective actions.

The same applies (even more) to software process improvement [Conradi and Fuggetta, 2002]. For instance, CMMI does not appear as suitable for smaller organisations. The overhead generated by the deployment of such a software process improvement framework may be regarded as too heavy.

As a heavy process, quality assessment should therefore be oriented towards more flexible approaches, that is, a way to tailor or customise the heaviest approaches in order to make them compliant with smaller contexts without dramatic overhead.

Similarly, provided that quality assurance is a demanding process requiring acceptance and collaboration from many involved stakeholders, quality assessment methodologies should also ensure that each effort of quality assessment will be meaningful and therefore not result in a waste of time, effort or money.

Chapter 3

Conceptualisation of the domain

Terminology and ontology of Software Quality

As we have seen in Chapter 1, Software Quality is a very broad field in which much research has been carried out. As a result, the domain of interest of this dissertation is conceptually rich and complex. Therefore, this chapter intends to narrow and structure this conceptual complexity in order for the remainder of the dissertation to build upon. This conceptualisation effort is divided in two parts. First, we provide a terminology of relevant concepts the remainder of the dissertation will refer to. Then, we provide an ontology that has been designed to structure these concepts and how they relate to each other. The terminology and the ontology rely on diverse source material found in the Software Engineering literature. Regarding Software Measurement, the terminology builds on previous efforts aiming to align different terminologies of measurement such as [García et al., 2006] , [Habra et al., 2008] or [Abran, 2010]. Other sources (e.g., ISO or IEEE standards) were used to provide specific definitions for some terms. This additional material is referenced throughout the chapter.

3.1 Terminology

This section provides the definition of several core concepts related to Software Quality. Any future occurrence of those terms are used accordingly to the definition found in this section, unless otherwise stated.

Definition 3.1 (Entity).
Any distinguishable object in the empirical world for which a measurement can be applied [Habra et al., 2008].

Additionally, [Fenton and Pfleeger, 1998] refines the notion of *object in the empirical world* and distinguishes three kinds of entities: products, processes and resources. Products and resources may be assimilated to **deliverables**, as defined within de CMMI framework. An entity is characterised by a set of **attributes**. Examples of software entities to be measured include a piece of code, a design artifact, a database, a programming task, a maintenance process or any other intermediate software product or process [Habra et al., 2008].

Definition 3.2 (Entity class).
The collection of all entities that satisfy a given predicate. [García et al., 2006]).

Definition 3.3 (Entity population).
A set of empirical entities having similarities [Habra et al., 2008].

The last two terms are closely linked. As matter of fact, an **entity class** characterises an entire **entity population** at once.

Definition 3.4 (Attribute).
A property of an ***entity*** *that can be determined quantitatively, i.e., which a magnitude can be assigned to [Habra et al., 2008].*

The concept of attribute is also defined as *a measurable physical or abstract property of an entity* in IEEE Standard 1061 [IEEE, 1998]. The additional information given by this second definition is relevant in the context of this work since many resources are in fact abstract constructs.

Definition 3.5 (Base attribute).
A base attribute is a simple property defined by convention, with no reference to other attributes, and possibly used in a system of attributes to define other attributes [Habra et al., 2008].

Definition 3.6 (Derived attribute).
A derived attribute is a property defined in a system of attributes as a function of base attributes [Habra et al., 2008].

Definition 3.7 (External attribute).
An attribute that can only be measured with respect to how the entity relates to its environment [Habra et al., 2008].

Definition 3.8 (Internal attribute).
An attribute that can be measured purely in terms of the entity being measured [Habra et al., 2008].

The distinction between internal and external attributes is specific to the

ISO/IEC 9126 standard. Internal attributes may be evaluated statically (i.e., without any execution of the software system and therefore without interference from environmental factors). External attributes must be evaluated dynamically (i.e., during the execution of the software system and with regard to its environment).

Definition 3.9 (**Measure**).
The number or category assigned to an attribute of an entity by making a measurement [ISO/IEC, 2001a].

Measure is also a synonym for **measurement value**.

Definition 3.10 (**Base measure**). *Measure defined in terms of an attribute and the method for quantifying it [SEI, 2010].*

A base measure may be seen as a **measure** defined for a base attribute. Base measure is also a synonym for the metrologic concept of base quantity [ISO/IEC, 2007b].

Definition 3.11 (**Derived measure**). *Measure that is defined as a function of two or more values of **base measures** [SEI, 2010].*

A derived measure may be seen as a **measure** defined for a derived attribute. Derived measure is also a synonym for the metrologic concept of derived quantity [ISO/IEC, 2007b].

Definition 3.12 (**Measurement**).
*The characterisation of an **attribute** in terms of number and symbols [Abran, 2010].*

This definition adopts a general point of view in order to stay consistent with most definitions found in the literature. Also defined as *"a set of operations to determine the **value** of a measure"* in [SEI, 2010] and as *"the use of a **metric** to assign a **value** (which may be a number or category) from a **scale** to an **attribute** of an **entity**"*, according to [ISO/IEC, 2001a].

Definition 3.13 (**Measurement function**).
An algorithm or calculation performed to combine two or more base or derived measures [García et al., 2006].

Definition 3.14 (**Measurement life-cycle**).
The whole process of measurement involving the design of measurement methods, the application of measurement methods and the exploitation of the measurement results [Habra et al., 2008].

Definition 3.15 (**Measurement method**).
A logical sequence of operations, described generically, used in the performance of measurement [ISO/IEC, 2007b].

Definition 3.16 (**Measurement procedure**).
A set of operations, described specifically, used in the performance of particular measurements according to a given context [ISO/IEC, 2007b].

Definition 3.17 (**Measurement plan**).
A plan that specifies and organises the step of the measurement and specifies why and what to measure, how to measure and who is responsible for the measurement (adapted from [Briand et al., 1997b]).

In other words, the measurement plan specifies the goals of the measurement, what **measurement values** to collect, through which **measurement procedures** and from which **entities**. It also specifies who is supposed to perform these activities.

Definition 3.18 (**Metric**).
*The defined **measurement method** and the **measurement scale** [ISO/IEC, 2001a].*

Also defined as *"a function whose input are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality"* in [IEEE, 1998].

Definition 3.19 (**Information need**).
Insight necessary to manage objectives, goals, risks, and problems (ISO/IEC 15939 definition from [García et al., 2006]).

Definition 3.20 (**Decision criteria**).
Numerical thresholds or targets used to determine the need for action or further investigation, or to describe the level of confidence in a given result [Abran, 2010].

Definition 3.21 (**Analysis Model**).
An algorithm or calculation combining one or more base and/or derived measures with associated decision criteria (ISO/IEC 15939 definition from [Abran, 2010]).

In other words, the analysis model bridges the gap between pure measurement (i.e., values that convey a neutral information) and actual quality assessment (i.e., values that convey a meaning that supports the decision-making process). Analysis models are defined in order to produce an **indicator** that can be interpreted with regard to a set of **decision criteria**.

Definition 3.22 (**Indicator**).
A measure providing an estimate or evaluation of specified attributes derived from a model with respect to defined information needs.

An indicator is produced by an **analysis model** and interpreted according to a set of **decision criteria**. Fundamentally, an indicator is an evaluation of a **derived attribute** that conveys a interpretable meaning, regarding the quality goals in a specific context. Indicators support the decision-making process and the communication with **stakeholders**.

Definition 3.23 (**Quality factor**).
A condition or characteristic which actively contributes to the quality of the software. [Mccall et al., 1977]

The definition of quality factor provided here is compliant with [IEEE, 1998] that defines it as *"a management-oriented attribute of software that contributes to its quality"*. Quality factor is a synonym for quality characteristic defined in [ISO/IEC, 2001a]. In the remainder of this dissertation, the term quality factor will be used as the general term to refer to a software attribute defined to assess quality, regardless of the specific term used in a given quality model (e.g., quality attribute in CMMI, quality characteristic in ISO/IEC 9126, etc.).

Definition 3.24 (**Quality model**). *The set of characteristics and relationships between them, which provides the basis for specifying quality requirements and evaluating quality [ISO/IEC, 1999].*

Additionally, [Deissenböck, 2009] defines quality models as *"structured collections of criteria for the systematic assessment of an **entity**'s quality"*.

Definition 3.25 (**Scale**).
A structured set of values associated with an attribute which is used to compare different entities according to that attribute [Habra et al., 2008].

The scale of a given attribute and guaranteed by its evaluation method is essential in order to inform the measurer about the kind of modification the measure can undergo in any subsequent processing. Five types of scales are used in software measurement: nominal, ordinal, interval, ratio and absolute. They are defined as follows [Fenton and Pfleeger, 1998]:

Definition 3.26 (**Nominal Scale**).
Nominal scales only provide a classification of data with arbitrary labels and no ordering (e.g., repartition of each people in a group into two categories: Male or Female). Accept both numeric and non-numeric values.

Definition 3.27 (Ordinal Scale).
Ordinal scales provide an ordered classification of data where the distance between values is not important (e.g., restaurant ratings). Accept both numeric and non-numeric values.

Definition 3.28 (Interval Scale).
Interval scales provide an ordered and constant classification of data with no natural zero and where the distance between values is meaningful (e.g., temperature). Accept only numeric values.

Definition 3.29 (Ratio Scale).
Ratio scales provide an ordered and constant classification of data with a natural zero (e.g., height). Accept only numeric values.

Definition 3.30 (Absolute Scale).
Absolute scales are ***ratio scales*** *that only allow the identity transformation. This type of scale only allows the count of occurrences of an element or event.*

Definition 3.31 (Unit of measurement).
A scalar attribute of an entity, defined by convention, with which other attributes of the same type are compared in order to express their magnitude [Habra et al., 2008].

Definition 3.32 (Value).
The magnitude assigned to an attribute of an entity represented by a number and a reference [ISO/IEC, 2007b].

Definition 3.33 (Deliverable).
An item to be provided to an acquirer (i.e., a stakeholder that acquires or procures a product or service from a supplier) or other designated recipient as specified in an agreement [SEI, 2010].

This definition encompasses a broad scope of items, such as documents, hardware items, software items, services, or any type of work product.

Definition 3.34 (Process).
A set of interrelated activities, which transform input into output, to achieve a given purpose [SEI, 2010].

This definition of process is consistent with the definition of process provided in [ISO/IEC, 2008]

Definition 3.35 (Process measurement).
A set of operations used to determine values of measures of a process and its

resulting products or services for the purpose of characterizing and understanding the process [SEI, 2010].

Definition 3.36 (Project).
A managed set of interrelated activities and resources, including people, that delivers one or more products or services to a customer or end user [SEI, 2010].

Definition 3.37 (Stakeholder).
A group or individual that is affected by or is in some way accountable for the outcome of an undertaking [SEI, 2010].

In the remainder of this dissertation, we rely on this general description while limiting its scope to the quality-related aspects of the software development lifecycle. That is, the term stakeholder is used to refer to any individual (or group) that is either responsible for the quality assessment or relying on the results of quality assessment in any way. Therefore, the stakeholder population, as we define it, includes : measurement users [Abran, 2010] or metric customers [Westfall and Road, 2005], customers (i.e., *"the party responsible for accepting the product or for authorizing payment"* [SEI, 2010]), end users (i.e., *"a party that ultimately uses a delivered product or that receives the benefit of a delivered service"* [SEI, 2010]), etc.

Definition 3.38 (Software product).
The set of computer programs, procedures, and possibly associated documentation and data [ISO/IEC, 2008].

3.2 Ontology

This section intends to structure and formalise the relationships between the concepts defined in Section 3.1. The Software Quality ontology described below has been built to model these relationships. Additionally, the ontology aims to provide the conceptual foundation for the approach proposed in Part II of this dissertation. The process of building followed the steps of the ontology building proposed in [Uschold and King, 1995], which are the following:

1. Identify purpose
2. Building the ontology
3. Evaluation
4. Documentation

However, due to the nature or our purpose and the building process we applied, the third and fourth steps have not been carried out in a formalised way.

3.2.1 Purpose

As explained above, the general purpose of the Software Quality ontology is to structure the domain and to provide the conceptual foundations for the approach proposed in Part II. This general purpose translates as a more concrete goal: align existing Software Measurement ontological structures and integrate the terminology of Section 3.1 into this aligned ontology. The other specific goal is to use this ontology to investigate how declarative and analytical approaches are related to each other and how we may provide an integrated view of those two paradigms.

The ontology is thus not an end but a support for the remainder of the dissertation. Additionally, it builds on a body of knowledge that is already well established. Therefore, the process calls for less validation and documentation.

3.2.2 Building the ontology

The first step of the building process is the ontology capture step, which is defined as [Uschold and King, 1995]:

1. identification of the key concepts and relationships in the domain of interest (i.e., scoping)
2. production of precise unambiguous text definitions for such concepts and relationships
3. identification of terms to refer to such concepts and relationships

In order to identify the key concepts (and provide the definitions) to include in the ontology, we relied on the terminology provided in Section 3.1. Regarding the structure of the ontology (and the name of the relations), we align the structures proposed in [Chirinos et al., 2005] and in [García et al., 2006], as well as the Measurement Information Model defined in [ISO/IEC, 2007a].

3.2.3 Evaluation and documentation

As explained above, the purpose of the ontology does not call for a heavy validation or documentation process. Our efforts regarding the evaluation mainly consisted in ensuring that the definition and the relationships of the concepts contained in the ontology were compliant with any of the definition and relationships provided by the multiple sources used to capture the concepts. Although its purpose is more operational than definitional, we also took into account the conceptual model proposed in the GQM/MEDEA approach [Briand et al., 2002] and verified that no conflict existed between the two models (i.e., that no element of the GQM/MEDEA conceptual model is incompatible with any concept of the ontology) .

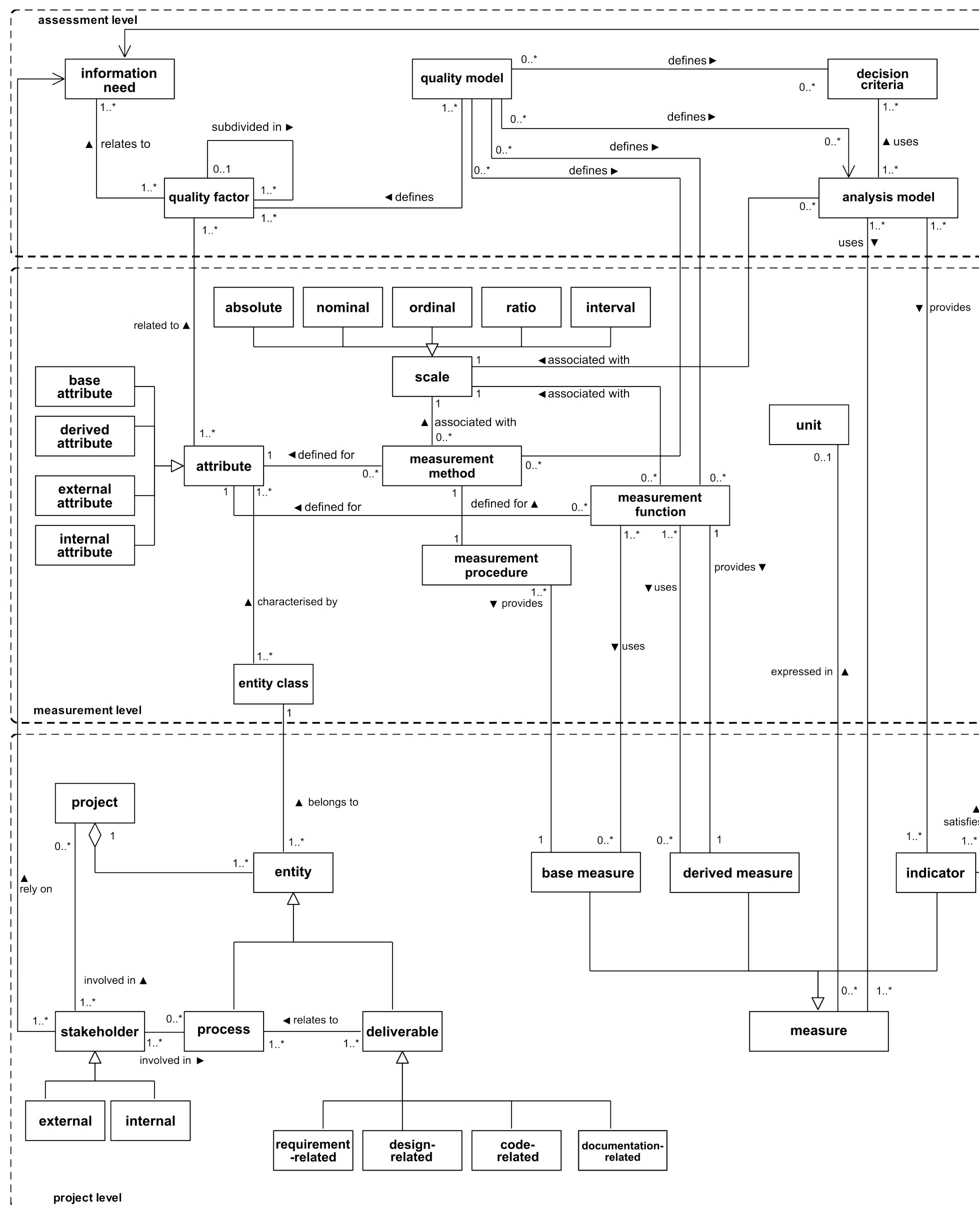

Figure 3.1: The software quality assessment ontology

The documentation of the ontology mainly consists in the terminology provided in Section 3.1 and the remainder of this chapter.

3.2.4 Software Quality ontology

Figure 3.1 shows the resulting Software Quality ontology, expressed as an UML class diagram. The ontology slightly extends the scope of the Software Measurement ontology proposed [García et al., 2006] in order to integrate project-related

concepts mostly inherited from the CMMI framework. In comparison to the Model for Software Measurement proposed in [Chirinos et al., 2005], it provides less focus on the mathematical concepts of Software Measurement. The same applies to the GQM/MEDEA conceptual model [Briand et al., 2002], which provides more focus on the operational aspects of the measurement process (e.g., corporate objectives, tactical goals, etc.) that we did not include at this stage.

The ontology itself distinguishes 3 levels: assessment level, measurement level and project level. The first two levels regroup concepts with an upper level of abstraction that relates to the process of defining measures and indicators. The last level displays concepts with lower level of abstraction, representing actual resources and actual values. This division reflects the fact that quantitative approaches to quality assessment are actually processes involving two separate phases with different scope in terms of abstraction. Quantitative approaches require a conceptual step where all the relationships between measures and their meaning in terms of quality are defined and a subsequent operational step where measurement is performed and the conclusions about quality are drawn thanks to the implicit model developed in the conceptual level.

Assessment level

The assessment level encompasses concepts that we classified as goal-oriented and stakeholder-oriented. The concepts at this level relate to the structuring of a series of goals that the project has to satisfy to, and how the level of satisfaction will be monitored on the basis of quantitative data defined at the measurement level. It also describes how declarative approaches and analytical approaches interact to provide the information required from the stakeholders.

As shown in Figure 3.1, the central concept is the **quality model** that defines a set of **quality factors**. These factors may be structured hierarchically and bridge the gap between an **information need** a **stakeholder** relies on and measurable concepts (i.e., **attributes**). **Quality models** may define **decision criteria**, **analysis models**, **measurement functions** and **measurement methods**, although it is not always the case, as we have seen in Chapter 1.

Analysis models rely on **decision criteria** and **measures** in order to produce an **indicator** that will respond to specific **information needs**.

Measurement level

The measurement level regroups concepts that we classified as purely related to the quantification process, that is, the association of a given magnitude to given properties of project-related elements. This distinction is mainly based on the fact that this magnitude (or value) is a neutral information that does not convey a quality information as-is (e.g., the size of an element). These concepts are mainly inherited from the representational measurement theory.

Figure 3.1 shows that the fundamental concept of the measurement level is the **attribute**. Attributes may be **base attributes** or **derived attributes** and may be **internal** or **external**. Although the tendency in the source material used to build the ontology is to associate **attribute** and **measures**, we chose to put the **measurement methods** and **functions** at the centre of this relationship. This choice is justified by our will to clearly separate the measurement level and the more concrete project level. Therefore, we can formalise that **measurement methods** are defined for **(base) attributes** and **measurement functions** are defined for **(derived) attributes**. Similarly **scales** are linked to **methods** and **function** instead of **measures**, in order to emphasise the fact that the adequate definition of the **methods/functions** is crucial to provide a **measure** within the desired **scale**. **Units** may be defined at this level and are associated to a specific **measure**, in order to emphasise the fact that **units** are closer to the actual value, acting as a type characterizing the value. **Measurement procedures** are also defined at this level, bridging the gap between the conceptual **measurement method** and the actual measurement value.

At this level, the entities exist as **entity classes**, that is, generic descriptions that characterise an entire entity population. Those **entity classes** may be characterised by a set of **attributes**.

Project level

The project level focuses on the concepts that represents the more operational level of the quality assessment process. The concepts regrouped at this level characterise people involved in the project, the quantitative information they can consult and the concrete elements characterised by this quantitative information. Besides, this part of the ontology provides a basic typology of relevant quantifiable elements, based on the review of existing measurement approaches provided in Chapter 1.

As shown in Figure 3.1, from a measurement perspective, the **project** is an aggregation of **entities** that may be **deliverables** or **processes**. The two subtypes of **entities** are interrelated (i.e., **deliverables** may be used as input for a **process**, and a **process** should result in one or more output **deliverables**). Additionally, the deliverables may be classified as **requirement-related**, **design-related**, **code-related** or **documentation-related** elements. Although this typology may be refined, it provides a sense of the vast scope of the procurable quantitative information.

The **stakeholders** have also been classified according to their status. All **stakeholders** are associated to one or more **information needs** and may be involved in the various **processes** of one **project**. **External stakeholders** (i.e., customers, end-users, higher management, etc.) are generally involved in a more indirect way (e.g., requirements elicitation, providing information on the

domain in which the development takes place, etc.). On the other hand, **internal stakeholders** (i.e., designers, programmers, etc.) are the main actors of the **processes** in which they are involved.

The quantitative information at this level is comprised of three types of **measures**: **base measures**, **derived measures** and **indicators**. The **information needs** of the **stakeholders** are satisfied by **indicators**, produced by an **analysis model**. **Analysis models** may rely on any type of **measures** in order to produce their output. **Derived measures** may be computed thanks to **base measures** or **derived measures**, through a **measurement function**. Finally, **base measures** may only be acquired through the application of **measurement procedure** targeting a specific **entity**.

Part II

Model-Centric Quality Assessment

Chapter 4

Overview of the approach

Theoretical foundations and MoCQA framework

4.1 Objectives of the approach

As explained in Chapter 2, Software Quality still suffers from several shortcomings and raises issues that researchers strive to address. Some of these concerns (such as the need for empirical validation of measurement methods, improved agreement on what attribute a given measure actually measures, etc.) cannot be addressed globally. These concerns have to be studied separately and will find answer as Software Quality matures as a field.

However, other issues (such as the organisational issues, the need for better communication between involved stakeholders, the need for processes allowing quality assessment to adapt to a specific context, etc.) may be solved through the development of more adequate methodological constructs.

In order to address those concerns, we propose a theoretical approach to quality assessment that relies on three specific core notions. These notions are either new in the field of quality assessment or already known notions that have been revised. The introduction of those three notions impact how quality assessment has to be performed. Therefore they require to adapt the quality assessment methodology but are expected to provide several benefits that will help solve or at least alleviate the above issues. These three notions are the following:

- Model-driven quality assessment
- Explicit and integrated quality assessment modelling
- Dedicated quality assessment life-cycle

The remainder of this section details each of these 3 notions.

Model-driven quality assessment

This notion is not entirely new in software quality assessment. In many regards, goal-driven measurement methods based on the Goal/Question/Metric approach define a model (although it is somewhat implicit) of the measurement that must be performed. That measurement model guides the quality assurance and this approach is therefore model-driven. We propose to push this function of goal-driven measurement models further by introducing *quality assessment* models designed to provide useful information to the different members involved in the development team as a reference regarding quality goals and related efforts.

As for goal-driven measurement models, the main objective of *quality assessment models* is to assist the quality assurance team in the planning and execution of the quality assessment process for a specific development context. The models are therefore designed to record information on the quality goals and the evaluation methods that are to be used. However, quality assessment models are also designed to record more specific information on the resources the development team is acting on and to relate these resources to the high-level quality requirements identified in the context. In order to help communicate with the managers or end users, quality assessment models also record information on the way high-level quality indicators should be interpreted and which actions should be taken in the software development process according to these interpretations.

Recording this heterogeneous information in a central model and using this model as a basis for the measurement process is the core of model-driven quality assessment. It pursues the goal of ensuring that the quality assessment performed is meaningful regarding the needs of all the relevant stakeholders.

Through quality assessment models, the quality assessment becomes a full-fledged model-driven process. Model-driven engineering relies on design-related models used through the implementation process to provide a software product all stakeholders can agree upon. In the same way, our approach proposes to rely on a quality assessment model that will result, through the process of applying the measurement plan, in a quality profile that represents the quality-related requirements and their current state of satisfaction, while granting a common understanding of these elements among the various stakeholders.

Explicit and integrated quality assessment modelling

Explicit quality assessment modelling denotes the fact that, in addition to different types of information required to support an efficient model-driven quality assessment, this information has to convey enough elements to make it useful to the stakeholders. The focal point of explicit quality assessment modelling is the expressiveness of the information the quality assessment model records. This expressiveness of the model in terms of what concepts it is able to model is a key to the efficiency of the process.

Integrated quality assessment modelling means that the information contained in the model has to include all relevant quality-related elements within the same model so that all the effort regarding the quality assessment process is available in a centralised way. Consequently, the quality assessment model should not focus on a single product but take into account the set of all relevant artefacts produced during the development into account. This also implies that the quality assessment model should support the integration of quality factors from multiple quality models and measurement/estimation from multiple sources.

More concretely, in order to ensure the usefulness of model-driven quality assessment, quality assessment models have to demonstrate two additional features: they have to be operational and customised.

A *customised* quality assessment model is specifically designed for a given software development context, thanks to the merging of relevant quality factors from multiple quality models and adequate measurement methods to evaluate them. Instead of deploying specific quality models for various software products, the customised quality assessment model models all quality assessment efforts planned for the entire project. The design of such a model requires a mechanism to align and convert the integrated elements and a means to keep track of the various sources used in the same quality assessment model. Additionally, the language used to express the quality assessment model must provide enough constructs to express and describe the resources available in the context in order to define them as measurable entities and/or points of improvement.

In order to become *operational*, the quality assessment model must ensure that each stakeholder has all the information she requires to perform the task she is expected to achieve in the context of quality assessment

The quality assurance team has to be able to perform the evaluation (measurement/estimation) and assessment (producing indicators) based on the quality assessment model alone (without a heavy operationalisation process). The manager and/or end users have to be able to understand the quality profile and interpret it in an adequate way and in the same way the quality assurance team does. Finally, the development team has to rely solely on the *operational* quality assessment model in order to be aware of the implication of their current task in terms of software quality. In other words, they have to be aware of the impact a modification of the current resource they are working on will have on the overall quality of the project, and which other resources are involved in this relationship.

The concept of *operational customised quality assessment models* is therefore the basic construct that will support the notion of explicit and integrated quality assessment modelling. Its goal is to guarantee that all that is needed to perform the quality assessment in a sound way is available.

Dedicated quality assessment life-cycle

Software development and quality assessment are often seen as separate activities and therefore, the two processes remain more of less independent from each other.

While the notion of measurement life-cycle (see Section 3.1) already exists, this life-cycle remains separated from the software development and does not really coincide with the various stages of the development life-cycle.

Besides, during the past two decades, software development processes have seen many efforts to improve their effectiveness. New paradigms have emerged (i.e., model-driven engineering, Agile methods, etc.) and have altered the way software development is conducted. Software development cannot be regarded as a straightforward waterfall-like process anymore and the way quality assessment is performed should follow this evolution.

In consequence, a quality assessment life-cycle should be envisioned with a broader scope, as a process that follows closely the software development life-cycle and helps adapt the quality assessment process to the requirements of a particular stage of the development. The approach we propose postulates that software development and quality assessment life-cycles are parallel activities that impact on each other and therefore should be performed simultaneously.

In order to fit the context of a given software development project, the supporting operational customised quality assessment model has to be designed and refined in parallel to the products themselves. Consequently, it defines its own life-cycle that must deal with its own decision-making process.

Thanks to the introduction of these core notions, the approach to quality assessment we propose is expected to provide the following benefits:

- help plan and adjust the quality assessment process throughout the software life-cycle;
- provide a quality assessment that fits the specific context in which it is performed;
- help detect the flaws in software measurement methods that are used;
- improve the overall acceptance of quality assurance activities;
- improve the communication between stakeholders in order to ensure that all of them are aware and understand the different goals regarding quality;
- improve the awareness of quality concerns among the various stakeholders in order for them to converge towards said goals.

Indeed, while model-driven quality assessment as we envision it provides a central mechanism (i.e., the operational customised quality assessment model) for each stakeholder to refer to, explicit and integrated quality assessment modelling ensures that the information contained in that model is not exclusively

useful to the quality assurance team. Besides, the quality assessment model provides enough measurement-related information to detect the weaknesses of the current assessment strategy. Finally, the quality assessment life-cycle allows the refinement and improvement of the quality assessment process as the software development unfolds.

4.2 Founding principles

In order to implement the core notions described in the previous section, the approach builds upon several related principles inherited from other fields of software engineering as well as from software quality and software measurement. The remainder of this section describes those principles and how they contribute to the foundation of the approach.

4.2.1 Constructivism

Constructivism is a concept inherited from learning theory. [Jonassen, 1991] explains that constructivism is a way to envision knowledge that opposes to the more traditional objectivist view. It postulates that knowledge is constructed by the knower based on mental activity while objectivism envisions knowledge as a pre-existing truth that is obfuscated from the knower who must strive to discover it.

These two different viewpoints on what knowledge is and how one acquires specific knowledge (i.e., constructivism versus objectivism) typically applies to software quality. Most declarative and analytic approaches (i.e., most quality models and metrics) are implicitly based on the objectivist assumption that the quality level is a pre-existent property of the software product. This pre-existent software quality (resulting from the more or less efficient work of the development team) has to be revealed by the quality assurance team thanks to measurement and quality models. This explains partially why software measurement is mainly performed at the end of a development cycle and still remains a control activity.

The quality assessment approach described in this dissertation adopts a more constructivist approach to software quality. Quality is not seen as a pre-existent aspect of the software product that must be discovered through measurement. Instead, the approach envisions quality as a series of aspects that must be "instilled" in the software product during development and maintenance, while software measurement is a tool to monitor the level of achievement of this process. This principle is close to the notion of "shared vision" implemented in the CMMI framework, that is, *"a common understanding of guiding principles, including mission, objectives, expected behavior, values, and final outcomes, which are developed and used by a project or work group"* [SEI, 2010].

Consequently, our approach to quality assessment may be regarded as the construction (among stakeholders) of a common knowledge that represents the current level of quality. At the beginning of a project, each stakeholder has his own perception of the quality requirements for the project (and therefore possesses its own mental model of what quality is) and how these requirements are prioritized. For instance, the developers may believe that they should focus on time performance or memory use optimisation while managers want to accelerate the time-to-market and the end users expect the software product to be stable and user-friendly). Adopting a constructivist approach means that the approach aims to reconcile those mental models in order to construct a shared mental model of the quality expectations for the project. This way, all involved stakeholder can collaborate efficiently to "implement" the quality requirements of this mental model.

This principle supports transversally the core notions developed on Section 4.1. Operational customised quality assessment models are constructivist mechanisms by design. Their emphasis on recording the rationale behind any quality assessment effort denotes this effort to reconcile divergences among the stakeholders' perceptions. The fact that the models are built and refined all along the software development life cycle, through the elicitation of quality requirements from the various stakeholders, also contribute to this instillation of quality into the software product.

4.2.2 Iterative / incremental life-cycle

In Software Engineering, the past two decades have witnessed the emergence of an increasing amount of iterative and incremental approaches, the least of which being the Agile paradigm [Beck et al., 2001]. These approaches are now widely recognized as beneficial to a successful software development.

In software development, an iterative approach refers to a scheduling and staging strategy that allows rework of parts of the system. An incremental approach refers to a scheduling and staging strategy in which pieces of the system are developed at different rates or times and integrated as they are developed [Cockburn, 2006].

[Read, 2005] explains that one of the main advantages offered by iterative/incremental approaches, is to *"provide a way to ensure the correct focus throughout development by addressing areas of technical concern early on, developing the key features/requirements first, obtaining real-world/customer/user feedback on early releases, calibrating effort on an ongoing basis and enabling the technical solution to evolve and be adjusted with minimal overhead"*.

Another fundamental advantage of iterative and incremental approaches it to allow mistakes during the course of a process and their correction in a short frame of time [Cockburn, 2006]. Any activity relying heavily on a human processing is

prone to witness the apparition of mistakes but addressing them and correcting them as they occur help people learn from the mistakes and is a beneficial process overall. Booch [2004] refers to this process as "gestalt, round-trip design", emphasizing the human characteristic of learning by completing.

Despite an increasing level of automation witnessed in software metrics, quality assessment still remains a process that relies heavily on human processing (e.g., prioritization of the quality goals, definition of the corrective actions to undertake, etc.) [Briand et al., 2002]. As such, an iterative management of software quality may be beneficial. As a matter of fact, [Dromey, 1996] already shows that quality models should be refined gradually to fit the goals and the context they are used in. In order to support the notion of quality assessment life-cycle, this principle of successive iterations is crucial. The main hindrance to an early quality assessment is the fact that measurement plans often require a certain level of maturity in order to be applied. Relying on an iterative quality assessment process makes the integration of less sophisticated measurement/estimation methods possible during the early phases of the development. Then, the methods are refined as the evaluated product gains in maturity. Although the first iterations could integrate very rough and imprecise evaluation methods, they would at least provide indicators regarding the global direction in which the software quality is heading. On the other hand, addressing quality assessment through an incremental process let the quality assurance team avoid dealing with goals that are not yet clearly stated or measurable entities that are just not mature enough to undergo any relevant evaluation.

This principle complements the constructivist view adopted by the approach. If quality assessment is perceived as the construction of a common knowledge, then quality assessment also implies that miscommunication inevitably arises between stakeholders. Adopting an iterative/incremental approach to this construction of quality knowledge is the key to steady progresses towards the shared view of what quality means for a given software product. Following a constructivist approach means reconciling the various mental models or the stakeholders. As a consequence, relying on an iterative/incremental approach to the quality assessment life-cycle means *converging* gradually towards a common mental model.

4.2.3 Involvement of the stakeholders

This principle is one of the cornerstones of requirement engineering. It is recognized to be of crucial importance in order to lead a software development project to a successful conclusion [Sharp et al., 1999]. Consulting the stakeholders in order to understand their requirements regarding the product that will be developed is therefore a widespread practice.

This principle also applies in the field of Software Measurement. As a matter of fact, [Westfall and Road, 2005] states that identifying the customer for each

metric is the first step towards useful software metrics. The author defines a customer for a metric as *"the person (or people) who will be making decisions or taking action based upon the metric"*.Those customers come in very different types and have different objectives and needs. Besides, *"if a metric does not have a customer, it should not be produced. Metrics are expensive to collect, report, and analyse so if no one is using a metric, producing it is a waste of time and money."* [Westfall and Road, 2005]. In order to support the notion of explicit and integrated quality assessment modelling and allow the quality assessment life-cycle to be efficient, the involvement of the stakeholders is essential.

The proposed solution therefore integrates this principle by providing an assessment methodology that is built around the "metric customers" (referred to as stakeholders in this dissertation). The approach, allows them to define their specific quality-related information needs and *integrate them with the global quality goals for the development*. This requires the identification of the stakeholders and their requirements (as proposed by [Westfall and Road, 2005]). Besides, we also have to link the various quality goals to each other. Pushing this principle even further, stakeholders should be able to provide input to the quality assurance team at each critical step of the quality assessment process (i.e., after the evaluation in order to see if the evaluated quality seems to comply with their practical experience).

4.2.4 Goal-Driven definition of measures

The fact that any measurement process should adopt a top-down approach is well documented, beginning with the Goal/Question/Metric proposed in [Basili and Weiss, 1984]. A top-down approach consists in establishing clear goals prior to the definition of any measurement method. Conversely, a bottom-up approach to measurement definition consists in measuring every possible entity in the context through every possible measurement procedure available before any attempt to link these results to any high-level quality goal.

The top-down goal-driven definition of measures provides a more focused measurement plan and spares the effort and time that would be wasted on useless measurement result collection. This principle is therefore crucial to support the core notions of our approach. In consequence, the design of operational customised quality assessment models has to be carried out following a top-down process, allowing to relate measure to a specific goal in addition to its already associated stakeholder.

4.2.5 Ecosystemic viewpoint

As explained in Chapter 2, software engineering has started to extend its scope from pure software code to various elements surrounding the code itself. The

software development process involves several different products (i.e., design, documentation, etc.) that are linked together by complex relationships. In consequence, many research works focus on the software ecosystem in order to gain a better understanding of the environment in which the development occurs.

This principle also applies to our approach. In order to support the notion of integrated quality assessment modelling, products should be studied in connection with each other. Assessing a product in isolation from the others is likely to be inaccurate since most of the artefacts produced within the development are connected and interdependent. In order to provide a view that is compliant with the notion of ecosystems, the approach proposed in this dissertation operates at the *software project* level. Based on the definition of project provided in the CMMI framework (see Chapter 3), we define the software project as follows:

Definition 4.1 (Software project).
A collection of deliverables linked together by transformational processes and providing a collection of runtime features in order to satisfy a set of user requirements.

In the remainder of this dissertation, the term software project is used according to this definition. This definition reflects the fact that software is not considered as a black-box, but as a network of interconnected products displaying various levels of abstraction or maturity, associated with observable runtime features. Additionally, this definition of software project is fully compliant with the notion of software ecosystem (i.e., a collection of software artefacts and/or projects, developed and co-evolving in the same environment). Indeed, a software project could include various other software projects, perceived as separate deliverables.

This specific perspective impacts the way operational customised quality assessment models should record the information regarding the measured entities. According to the definition, a quality assessment model targets a subset of the software project, that is, a set of measurable entities that are part of a network of resources. The quality assessment model should offer the possibility to keep track of these relationships.

4.2.6 Definitional and analytic approaches integration

As shown in Chapter 1, many definitional (i.e., quality models) and analytical (i.e., measures) approaches to software quality have been proposed. The shift from a product-based approach to a more ecosystemic perspective (and the notion of integrated quality assessment modelling) involves the coexistence of various methods and quality factors within the same model.

This principle calls for an ontological support for the design of the operational customised quality assessment models. An adequate ontological support makes

it possible to align the various analytical and declarative methods in order to include them in the same quality assessment model while avoiding any conceptual mistakes.

4.2.7 Reusability

Reusability is widely believed to be a key to improving software development productivity and quality [Biggerstaff and Richter, 1989]. The advantage of reusability is that it solves two main problems inherent to software development. On the one hand, it helps spare time by avoiding duplicate efforts for similar tasks. On the other hand, reusability helps prevent making mistakes by avoiding to repeat something that has already been done.

These two issues are common to the ones encountered in quality assessment. Indeed, as we have seen in Chapter 2, the operationalisation is a non-trivial process that requires time and effort in order to be executed correctly. Similarly, we have seen in Chapter 1 that customisation is often required in order to adapt quality models with a very generic scope to a specific context. This process is not trivial either and requires some more effort and time. Reusing part of the effort provided in these activities would help spare time and effort.

In order to support the core notions from Section 4.1, reusability proves even more potent. Reusing an already customised hierarchy of quality goals stakeholders agree upon helps avoid miscommunication between stakeholders. This principle requires the operational customised quality assessment model to allow some level of modularity regarding the elements that have already been included. An already operationalised hierarchy of quality goals should be reusable as-is or with minor modification (i.e., we may reuse an existing quality hierarchy with a different set of metrics, provided that the metrics are compatible with the quality goals, etc.). It therefore requires a mechanism that goes beyond the static definition of quality models or measurement methods and allows to understand their intrinsic mechanisms in order to make them modular and, therefore, reusable. This principle also contributes to balancing the extra effort needed to apply the specifics of the approach (such as the involvement of the stakeholders, the modelling effort that must be provided, etc.).

4.2.8 Domain-specific languages and expressiveness

Domain-specific languages (DSLs) are programming languages or specification languages dedicated to particular problem domains, representations techniques or solution techniques. Conversely, general purpose languages (GPLs) may be applied to various problems, regardless of the specific domain of the problem (e.g., Unified Modelling Language).

Domain-specific modelling languages are a common way to improve the modelling process. They contribute to reducing the learning curve for the users and improve the communication of specific ideas between team members. Besides, DSLs are known to be enablers of reuse as explained in [Mernik et al., 2005].

DSLs can be a good support in the field of software measurement as shown in [Mora et al., 2008]. The notion of explicit and integrated quality assessment modelling requires an efficient way to accomplish the modelling tasks and therefore, a domain-specific support has to be envisioned in order to support our approach.

4.2.9 Human aspect of software quality

Among the several good practices that have been identified by researchers regarding software measurement (i.e., goal-driven top-down definition of measures, clear identification of the measurable entities, etc.), taking the human aspect of software measurement into account is a key element to our approach.

As explained in [Westfall and Road, 2005], software metrics (and quality assessment in general) affect people and people affect measurement, that is *"whether a metric is ultimately useful to an organization depends upon the attitudes of the people involved in collecting the data, calculating, reporting, and using the metrics"*.

[Westfall and Road, 2005] identifies specific guidelines to help decrease the lack of acceptance regarding any measurement programs. Those guidelines are the following:

1. Don't measure individuals
2. Never use metrics as a "stick"
3. Don't ignore the data
4. Provide feedback
5. Obtain "buy-in"

The first guideline acts as a caveat regarding the use of productivity measures. Although it is tempting to try to improve the overall development process by monitoring the individual productivity of the team members, productivity measures typically menace to disrupt the team work flow by overemphasising the individual. [Westfall and Road, 2005] adds: *"Remember that we often give our best people the hardest work and then expect them to mentor others in the group. If we measure productivity in lines of code per hour, these people may concentrate on their own work to the detriment of the team and the project"*. Ultimately, it is more efficient to address processes and products than individuals.

The second guideline implies that if a measure is used as a threat against an individual, the risk of this individual reporting false data (i.e., over-optimistic measurement results) increases. Measures should therefore be used to provide

support and result in a collaboration between the stakeholders that aim to provide a clear understanding of the current overall quality of the software project.

The third guideline recommends to value the measurement results that are provided. More specifically, the measurement results should be integrated in the decision-making process (regardless of whether they are positive of negative). If data are ignored, the measurement program will likely become less and less efficient, until it is simply abandoned. As explained in [Westfall and Road, 2005], *"if the goals we establish and communicate don't agree with our actions, then the people in our organization will perform based on our behaviour, not our goals"*.

The fourth guideline relates to the involvement of the stakeholders. Measurement programs should result in back-and-forth exchanges between the people who collect the data, the people who design the program and the people who define the quality goals. For one, this exchange helps maintain the motivation regarding the data collection (since it shows that the data is actually used). It also reduces the possible reluctance due the fact that people don't know what the measurement data is being used for. The knowledge and experience of the team members may also be integrated in the quality assessment process through a back-and-forth feedback. Besides, *"feedback on data collection problems and data integrity issues helps educate team members responsible for data collection. The benefit can be more accurate, consistent, and timely data"*.

Finally, the last guideline recommends to include the team members in the design of the measurement program itself, so that the feeling of ownership is enhanced and the overall acceptance of the program increased. In addition, *"the people who work with a process on a daily basis will have intimate knowledge of that process. This gives them a valuable perspective on how the process can best be measured to ensure accuracy and validity, and how to best interpret the measured result"*.

These guidelines ultimately summarize the core principle we propose to adopt regarding the human aspect of measurement in our approach: shift from a *control* paradigm to a *guidance* paradigm. Quantitative approaches to quality assessment should help the development team instead of controlling its members. By making available extensive information to all the stakeholders, the notion of explicit and integrated quality assessment modelling contributes to this shift.

We propose to further implement the principle through the adoption of an hybrid point of view for the operational customised quality assessment models. The ecosystemic viewpoint of the approach already guarantees that the relationships between measurable entities (i.e., implementation, documentation, refinement, etc.) are considered during the assessment. Allowing the definition of these relationships, that is, these *processes*, as measurable entities as well provides a way to assess products and processes in the same context. Although people are involved

in these processes, assessing the activity instead of the skills or productivity of the people should remove the issues linked to the perception of the quality assessment process. The shift from the control perspective to a guidance perspective of quality assessment is also implemented through the iterative and participative process of the approach.

4.3 The MoCQA framework

Introduced in [Vanderose et al., 2010], the Model-Centric Quality Assessment (MoCQA) framework is an implementation of the theoretical approach defined in Section 4.1. Concretely, this software quality assessment framework has been designed to integrate (as seamlessly as possible) the principles described in Section 4.2 to help plan and support quality assessment during software development, from the early stages of development to the maintenance and evolution processes.

At its core, the framework defines a quality assessment metamodel based on the conceptual level of the software quality assessment ontology described in Chapter 3. The quality assessment metamodel therefore captures:

- concepts inherited from traditional quality models
- concepts inherited from software measurement
- a generalized typology of measurable entities

The quality assessment metamodel provides support (that is, an abstract syntax) for the systematic and consistent design of **operational customised quality assessment models** (described in Section 4.1), specifically designed for a defined software project and its particular environment.

On top of its core quality assessment metamodel, the framework defines a dedicated assessment methodology designed to support a quality assessment life-cycle built upon the design, exploitation and evolution of the operational customised quality assessment models.

Through this assessment methodology, the framework provides the support needed to produce coherent and structurally valid operational customised quality assessment models. The approach also provides support for an effective use of measurement (i.e., a measurement that is tailored according to the goals of the stakeholders and focus on the satisfaction of their quality-related information needs).

4.3.1 MoCQA models

Operational customised quality assessment models are the central mechanism that supports the model-driven quality assessment process proposed by our theoretical approach. MoCQA models implements this concept within the framework.

The main goal of MoCQA models is to centralize the relevant information to support the quality assessment process. Once defined, a MoCQA model takes the role of a map that guides the execution of the quality assessment process and the subsequent exploitation of its results.

Concretely, MoCQA models aim at providing the required support thanks to the combination of :

- a hierarchy of quality goals specifically designed for a given development environment (i.e., taking into account the specific environmental factors of the software project and the quality requirements of its stakeholders);
- a set of customised measurement/estimation methods designed to monitor the level of satisfaction of the various quality goals;
- a structured and detailed definition of the resources targeted by the measurement/estimation methods, taking into account their relations to each other and the multidimensional nature of the software project (i.e., multiple levels of abstraction/maturity for the resources).

As such, MoCQA models actually implement the concept of quality assessment model. Contrary to traditional quality models defining quality for a specific product, a MoCQA model extends this limited scope by documenting all the relevant assessment-related aspects for a given project (i.e., what/how/why/for whom we measure and inspect different parts of the project).

The quality assessment metamodel that supports the design of MoCQA models has been conceived to allow the alignment, tailoring and integration of quality models and measurement/estimation methods coming from different sources (Figure 4.1). It therefore grants that MoCQA models are *customised* quality assessment models.

The quality assessment metamodel also supports the detailed characterization (i.e., relation between quality goals and stakeholders, status of a given measurement/estimation method regarding its validation, etc.) of the information contained in MoCQA models so that they may be regarded as *operational* quality assessment models.

Finally, the quality assessment metamodel defines an abstract syntax for MoCQA models. This abstract syntax facilitates the design and revision of the models. Therefore, MoCQA models are not set in stone and are bound to evolve during the software development life-cycle. The refinement and evolution of MoCQA models constitutes an adequate support for the notion of quality assessment life-cycle.

4.3.2 Model-Centric Quality Assessment methodology

The introduction of operational customised quality assessment models brings the quality assessment process conceptually closer to the model-driven engineering

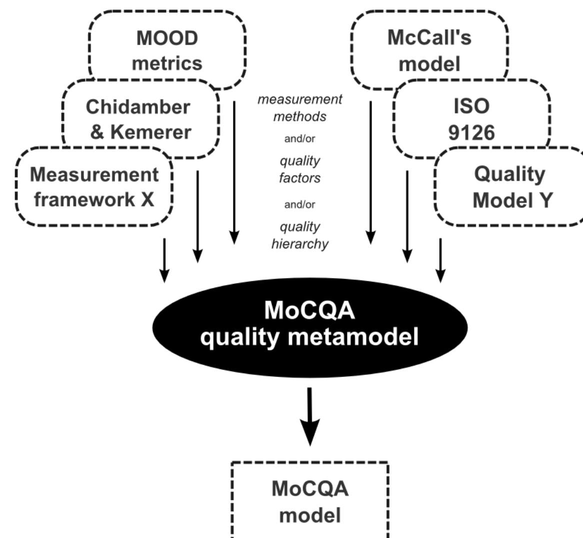

Figure 4.1: Integration of quality models and measurement/estimation methods

of the product itself (i.e., design of a model based on elicited requirements, "implementation" of the quality assessment process through the measurement plan, "testing" of the quality profile with regard to the needs of the stakeholders).

As a result, operational customised quality assessment models impact the way software quality assessment is performed. The main impact is the necessity to bind this conceptual model (mainly designed to communicate among stakeholders) to the actual (and possibly tool-assisted) measurement process. The second impact resides in the challenge of designing the model itself and acquiring the necessary knowledge from the stakeholders. Finally, the process involves a systematic reflection on the quality assessment process.

In consequence, the MoCQA framework introduces a specific methodology designed to support the use of MoCQA models. This methodology is the key to the implementation of the principles described in Section 4.2.

Among these principles, the goal-driven definition of measures can be found. The assessment methodology defined by the framework therefore implements the methodological principles of the Goal/Question/Metric approach [Basili et al., 1994]. It is thus mainly a top-down methodology instead of a bottom-up approach. It is therefore possible to map the steps of the MoCQA methodology with the steps of the GQM method to some extent.

However, the MocQA methodology deviates a little from the pure top-down approach. The main jeopardy of a top-down approach to measurement is to define a measurement plan that is not applicable in the end, due to the lack of specific entities to measure or the use of a measurement that is not applicable in the specific software development context. Therefore, the MoCQA methodology

allows the description of the measurable entities at hand before the quality goals are defined. The quality assessment metamodel makes that pseudo-bottom-up approach possible since it still requires the definition of specific goals and metrics for the measurable entities that have been considered to begin with. The process is still performed in a systematic way but allows more flexibility in order to adapt to a specific context.

In order to implement the iterative and incremental nature of the theoretical approach described in Section 4.1, the MoCQA methodology breaks the overall quality assessment process (or quality assessment life-cycle) into successive cycles, as defined hereafter:

Definition 4.2 (Quality assessment life-cycle).

Any number of quality assessment cycles (and resulting decisions) occurring in parallel to the software development and evolution life-cycle

Each iteration of the assessment methodology is thus called a quality assessment cycle, that we define formally as follows:

Definition 4.3 (Quality assessment cycle).

The sequence of quality-related activities beginning with the planning of the assessment and leading to the actual assessment of a software project. Each quality assessment cycle results in a set of decisions made by the development team about the forthcoming activities regarding the development life-cycle and the next quality assessment cycle.

The fact that the MoCQA methodology breaks down the process into iterative quality assessment cycles allow for a systematic revision of the quality goals (and quality model) and assessment methods. At the end of each cycle, the quality assurance team needs to reflect on the assessment performed so far and, together with the stakeholders, decide if the indicators and the way they are defined are relevant.

As shown in Figure 4.2, the quality assessment methodology defined by the framework decomposes each quality assessment cycle into five successive steps.

- **Acquiring contextual knowledge.** This step focuses on the elicitation of relevant contextual information on the software development environment and on the specific quality requirements.
- **Designing the MoCQA model.** This steps focuses on the creation and structural validation of a MoCQA model by instantiation of the quality assessment metamodel.

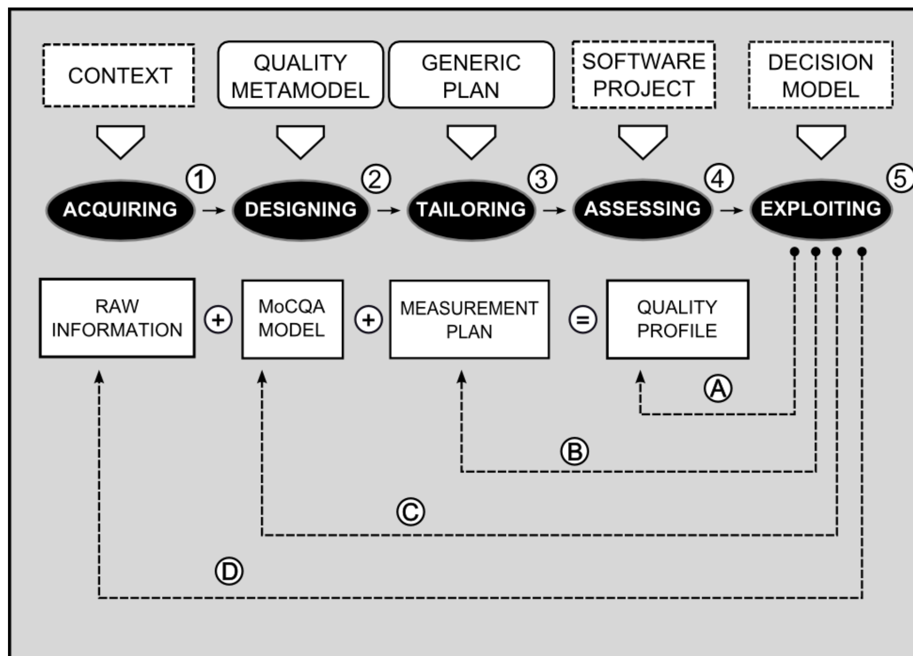

Figure 4.2: The MoCQA methodology

- **Tailoring of the measurement plan.** This step addresses the definition of practical guidelines for the measurement and quality assessment, based on the conceptual definitions provided in the MoCQA model.
- **Assessing the software project.** This is the step where the actual measurement-related and quality-related data (i.e., measurement results and indicators) are collected in order to produce a quality profile of the software project.
- **Exploiting the quality profile.** In this step the quality indicators are interpreted and used as input of the decision-making process related to the remainder of the development and/or the evolution processes and to the next quality assessment cycles.

Steps 3 & 4 are optional for every quality assessment cycle, as will be explained below.

The remainder of this Section provides more details on each of these steps.

Acquiring contextual knowledge

Input: The development context (i.e, stakeholders' knowledge, processes documentation, quality standards or norms the products/processes have to comply to, etc.).

Output: The raw information to be modelled in Step 2 as well as constraints that will be used throughout the quality assessment life-cycle.
Description: Ensuring that the quality assessment goals fit the specific development context is a key aspect of the approach. The acquisition of a relevant knowledge of this context is therefore the first step to perform.

Concretely, the acquisition step consists in listing all identified relevant environmental factors that are not directly targeted by the quality assessment metamodel (e.g., time constraints, budget constraints, constraints linked to development process, availability of resources, etc.). These factors may be useful during the exploitation step, especially in the decision-making process regarding the remainder of the life-cycle.

This step is also designed to involve the various stakeholders in the definition of the global assessment purposes in order to provide a quality assessment that is relevant for all of them.

Designing the MoCQA model

Input: The contextual information collected in Step 1 and the quality assessment metamodel.
Output: A MoCQA model designed to guide the remainder of the quality assessment cycle.
Description: Fundamentally MoCQA models are instances of the quality assessment metamodel defined by the framework. The process of designing a given MoCQA model can therefore be regarded as an instantiation process constrained by the raw information collected in the previous step. This design process, also referred to as *quality assessment modelling*, is the central task of this step of the assessment methodology.

Due to the fact that designing a MoCQA model represents the implementation of a GQM-like approach, this instantiation process must follow a specific order. Provided that MoCQA models can also be regarded as extended quality models, the guidelines defined to design a customised quality model in [Dromey, 1996] also apply to the design of a MoCQA model.

Tailoring the measurement plan

Input: The MoCQA model designed in Step 2 and the generic measurement plan implicitly defined by the framework.
Output: A measurement plan designed to perform the actual measurement and assessment of the software project.
Description: In essence, a measurement plan defines what measures have to be collected and how to identify them (i.e., identify/locate behaviour/resource X, Y, Z and apply measurement methods A, B, C to X, Y, Z). In many regards

MoCQA models themselves may be regarded as abstract measurement plans. This third step consists in the adaptation of this abstract measurement plan to make it operational, that is, providing actual guidelines (adapted to the actual environment) in order to allow the measurer to easily find the relevant measurable entities and apply the adequate measurement methods.

Concretely, in this step, measurement procedures are defined for each measurement method. Analysis models are structured and actual project resources (i.e., measurable entities) are located and tagged according to the corresponding measurable entity types defined in the MoCQA model. Other factors like the frequency of assessment are also defined in the measurement plan.

This step may be ignored if the planning of quality assessment is not yet complete and the aim of the current quality assessment cycle is to assess the MoCQA model itself.

Assessing the software project

Input: The measurement plan defined in Step 3 and the actual resources from the software project.
Output: A quality profile of the software project (i.e., a set of tagged resources/identified behaviours/activities, of measurement values, of quality indicators, and their relationships) that may be exploited in order to take actions regarding the software development process or the quality assessment life-cycle.
Description: During this step, measurement results are collected and the quality assessment is performed, according to guidelines of the measurement plan implementing the MoCQA model. This step consists in mapping the actual project resources with the MoCQA model through the measurement plan. This mapping translates in the analysis of the produced quality indicators on the basis of the predefined interpretation rules and with respect to the predefined scope of the quality goals.

This step may be ignored if the previous step has been ignored.

Exploiting the quality profile

Input: The quality profile produced in Step 4 and the decision model (that may be informal) provided by the stakeholders.
Output: A set of decisions regarding the actions that have to be performed, both in the context of the software development process and regarding the quality assessment life-cycle.
Description: This step brings the current quality assessment cycle to an end and is mainly concerned with the decision-making process based on the quality indicators (interpreted according to the rules defined in the MoCQA model). The decisions concern the continuation of the development life-cycle (i.e., what

improvements have to be performed, what parts of the project call for more investigation, what parts of the project may be considered satisfying).As such, this step also requires the participation of the stakeholders.

Due to the iterative nature of the process, the exploitation step also concerns the subsequent quality assessment cycles and how the assessment effort will be refined or augmented.

Regarding the quality assessment life-cycle, four main outcomes may arise from the decision-making process. Each of them will call for one of the allowed feedback loops described in Figure 4.2 and detailed hereafter:

1. The regular case is the reuse of the tailored measurement plan. The measurement and assessment step will likely be taken several times throughout the development in order to monitor the evolution of the quality indicators over time. Input data of the assessment models can help identify where effort should be consented to meet the given quality objectives. This case requires no conceptual redesign and redirect towards the measurement and assessment step (i.e, step 4 of the assessment methodology).
2. In some cases, the measurement plan may need to be adapted after important changes in the software project. For instance, a language migration would require the user to redefine the guidelines provided to identify the measurable entities. This case requires a light conceptual redesign and redirect towards the measurement plan tailoring step (i.e, step 3 of the assessment methodology).
3. Other cases will require the MoCQA model to be adapted after the apparition of a new quality-related information need or if flaws in the quality assessment process have been identified on the basis of the MoCQA model. For example, the developers could introduce the documentation into the software project and want to monitor its availability. The quality assurance team may also discover a better measurement method to evaluate a given attribute used in one to the analysis models. This case requires a heavy conceptual redesign and redirect to the MoCQA model design step (i.e, step 2 of the assessment methodology).
4. Finally, repeating the acquisition step (i.e, step 1 of the assessment methodology) could be necessary in some cases. The main reason for thoroughly involving the stakeholders once again in this process is to refine the quality profile and check if some quality goals have not been left out of the previous analysis. However, the interpretation of indicators and/or the actions that have been defined in the previous quality assessment cycle could raise controversy among the stakeholders and require to redefine them collectively to adapt the MoCQA model and improve the common understanding of quality for the project.

4.4 Structure

Each of the above methodological steps raise specific concerns and challenges. The next chapters focus on each steps of the methodology and elaborate on how the framework addresses those concerns.

- Chapter 5 focuses on the *design* of MoCQA models. It provides the complete specification of MoCQA models and a detailed description of the underlying quality assessment metamodel that supports their design. Concerns about structural coherence of the models are also addressed in this chapter.
- Chapter 6 explores the tasks required in order to collect the information during the *acquisition* step and provides an overview of possible methods to facilitate this process.
- Chapter 7 details the *tailoring* of the measurement plan. The topics addressed in this chapter include how the introduction of MoCQA models impacts the measurement process, consideration on how data models for the persistence of measurement values should be adapted and how MoCQA models may be enriched with metadata to bridge the gap between conceptual and operational levels of the quality assessment process. The *assessment* step is discussed in this chapter as well.
- Chapter 8 addresses the *exploitation* step and how a quality profile may be produced thanks to MoCQA models once the *assessment* step has been performed. This chapter also discusses how MoCQA models may be used to detect possible flaws in the quality assessment process, as well as be assessed and revised themselves, therefore improving the next quality assessment cycle.

Finally, Chapter 9 provides a description of the tool-support that has been considered and/or developed during this research in order to improve the operational effectiveness of the MoCQA framework.

Chapter 5

MoCQA models

Customised Operational Quality Assessment Models

As explained in Chapter 4, MoCQA models are the cornerstone of the Model-Centric Quality Assessment methodology. A MoCQA model is a model designed to support the model-centric quality assessment of a given **software project**. It records all the relevant information on the specific quality-assessment-related aspects of a given context. It is designed to guide the execution of a quality assessment cycle and the subsequent exploitation of its results.

MoCQA models implement the notion of *quality assessment model* described in Chapter 4. This concept differs from the notion of *quality model*, defined as a *"structured collections of criteria for the systematic assessment of an **entity**'s quality"* in [Deissenböck, 2009].

Indeed, contrary to quality models that define statically a set of quality factors for a software product and (possibly) the measures designed to evaluate them, a quality assessment model extends this scope and documents all relevant quality-related aspects to guide a quality assessment cycle, that is, the following:

1. a hierarchy of quality issues for the software project (i.e., quality goals characterised by the quality factor they embody, the part of the software project they are relevant for, the stakeholders they are defined for, the indicators used to assess how they are satisfied and the way these indicators should be interpreted.)
2. a definition of the quantification methods used to produce the indicators (i.e., the attributes that have to be evaluated and the definition of the measurement methods/functions used to evaluate them.)
3. a characterisation of the classes of entities to which the measurement has to be applied, as well as the relationships between these classes of entities.

In consequence, a formal definition of *quality assessment model* may be the following:

Definition 5.1 (Quality assessment model).
A structured collection of quality issues (associated with their indicators, measurement and/or estimation methods and related entity types), defined for the systematic assessment the quality of a software project.

Although it requires an additional effort in order to become fully operational, a quality assessment model (or MoCQA model) may therefore be regarded as a conceptual representation of a measurement plan (i.e., it describes what/how/why/-for whom we measure and inspect different parts of the project).

In order to implement the notion of explicit and integrated quality assessment modelling, MoCQA models require a dedicated ontological support that permits the integration of:

- qualify factors coming from diverse quality models to be embodied in quality issues (concepts of hierarchical quality models)
- measurement or estimation methods coming from different sources (concepts of software measurement)
- detailed descriptions of the entity classes that will be monitored (a generalized typology of measurable entities)

Besides, as explained in Chapter 4, MoCQA models have a dedicated lifecycle. They are designed on the basis of stakeholders' quality requirements, completed by the quality assurance team, refined and corrected as the software development process occurs. The ontological support has to allow the systematic and consistent design and refinement of successive versions of a given MoCQA model.

Therefore, this ontological support is provided by the framework in the form of a quality assessment metamodel. Designing a MoCQA model consists in instantiating this metamodel. The approach therefore follows the four-layer modelling procedure described in [Sprinkle et al., 2001] and shown in Figure 5.1. However, the lower level does not intend to provide a computer based system but a quality profile of the software project.

5.1 MoCQA models and Meta-Object Facility

In order to clarify the concepts of the MoCQA framework, this section describes how the core elements of our model-driven approach fit into the Meta-Object Facility (MOF) architecture [ISO/IEC, 2005a]. The Meta-Object Facility is the conceptual architecture supporting the Model Driven Architecture (MDA) de-

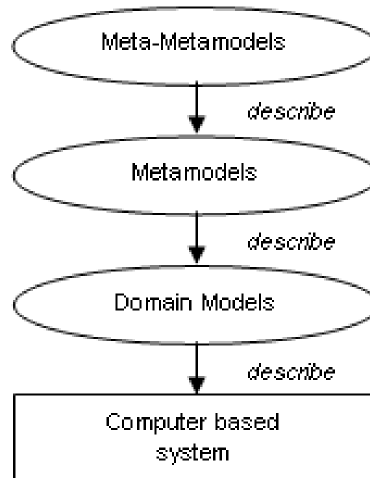

Figure 5.1: The four layers of modelling

fined the Object Management Group[1] (OMG) [Miller and Mukerji, 2003]. The MOF formalizes the four-layer approach shown in Figure 5.1 into four levels of modelling (from M0 to M3). These four levels allow the representation of concrete elements of the empirical world (e.g., a software system at level M0) through the definition of models (e.g., a UML class diagram at level M1) based on metamodels (e.g., the UML metamodel at level M2) which are themselves defined through a universal and auto-defined meta-metamodel (level M3) [OMG, 2006].

Figure 5.2 shows how the core elements of the MoCQA framework match this conceptual architecture.

The quality assessment metamodel logically fits at the **M2-level** and constitutes the origin of the quality assessment modelling process.

MoCQA models and there constitutive elements belong to the **M1-level**. They are therefore instances of the quality assessment metamodel. MoCQA models contain *definitions* of quality-assessment-related aspects, divided in three categories. First, the model contains the definition of quality issues (and related indicators, interpretations, analysis models). The model also provides the definition of measurement methods (and associated attributes, functions, scale and unit). Finally, MoCQA models defined at **M1-level** provide the specification of the types of entity that will be measured.

As part of the **M0-level**, the approach copes with *quality profiles.* The quality profile of a software project encompasses :

- the actual entities belonging to the entity populations defined by the entity types from **M1-level** models

[1] http://www.omg.org/

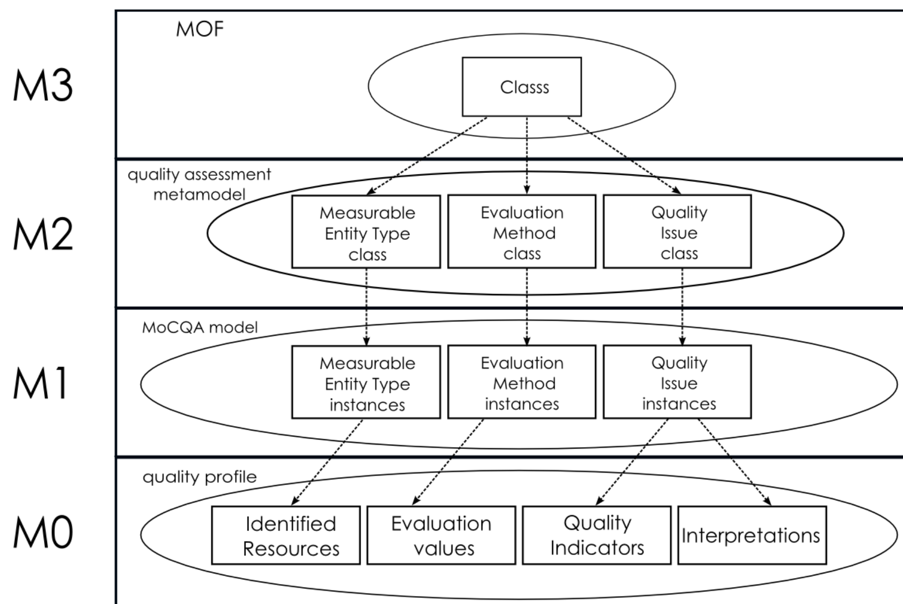

Figure 5.2: Multi-level hierarchy for the approach

- the actual measurement values collected from them through the measurement procedure defined for each measurement methods
- the quality indicators computed on the basis of the latter and a set of interpretation regarding the software project

Although these elements are common to many (if not all) quality assessment frameworks, the support offered by the MoCQA model is the fact that all of these **M0-level** elements are conceptually related thanks to the **M1-level** model. The relationships between the elements are therefore clearly stated and recorded. The quality profile is the mechanism that allows stakeholders to obtain answers to their information needs.

5.2 Quality assessment metamodel

This section details the **M2-level** of the MOF-based architecture described in the previous section. In consequence, this section focuses on the specification of the quality assessment metamodel that supports the MoCQA framework.

[Deissenböck, 2009] defines a quality metamodel as *"a model of the constructs and rules needed to build specific quality models"*. Based on this definition, we may propose the following formal definition for our quality assessment metamodel:

Definition 5.2 (**Quality assessment metamodel**).
A model of the constructs and rules needed to build specific quality assessment model)

The quality assessment metamodel has therefore to define all the concepts (and the relationships between these concepts) that may be included in a quality assessment model. In the context of our framework, it may also be considered an abstract syntax for MoCQA models. It ensures the robustness and coherence of MoCQA models' design and evolution.

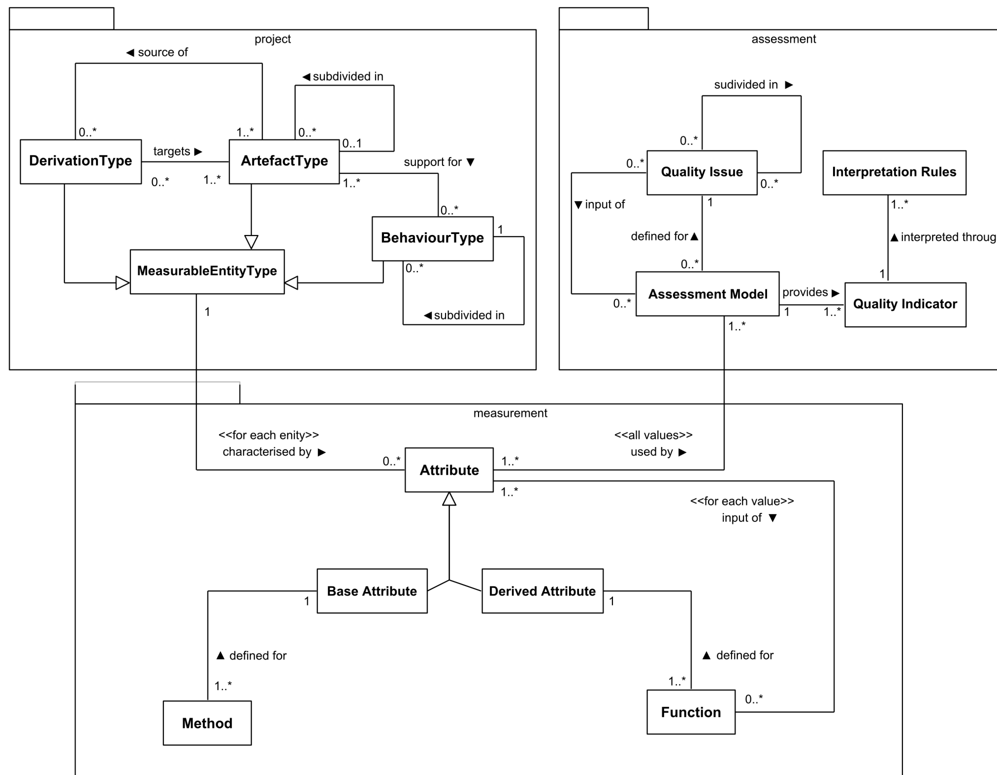

Figure 5.3: Simplified view of the MoCQA quality assessment metamodel

Figure 5.3 provides a simplified view of the MoCQA quality assessment model. This figure illustrates the concepts (or constructs) available to design a MoCQA model and how they may be associated. The attributes available to characterise each of the constructs will be detailed in the remainder of this section.

As explained in the previous section, MoCQA models are built upon three distinct components, each of them modelling a different aspect of quality assessment. Any coherent and complete MoCQA model should displays these three components. The concepts included in the quality assessment metamodel are therefore regrouped in 3 distinct packages:

The **assessment package** defines the constructs dedicated to the structured definition of quality issues and their indicators. This package is related to the assessment-level of the software quality ontology described in Chapter 3.

The **measurement package** defines the constructs dedicated to the specification

of underlying measurement and/or estimation methods that provide the input for the assessment. This package is related to the measurement-level of the software quality ontology described in Chapter 3.

The **project package** defines the constructs dedicated to the modelling of the entity types that will be measured or estimated. This package is related to the project-level of the software quality ontology described in Chapter 3, although it is not a direct translation, as we will explain in Section 5.2.1.

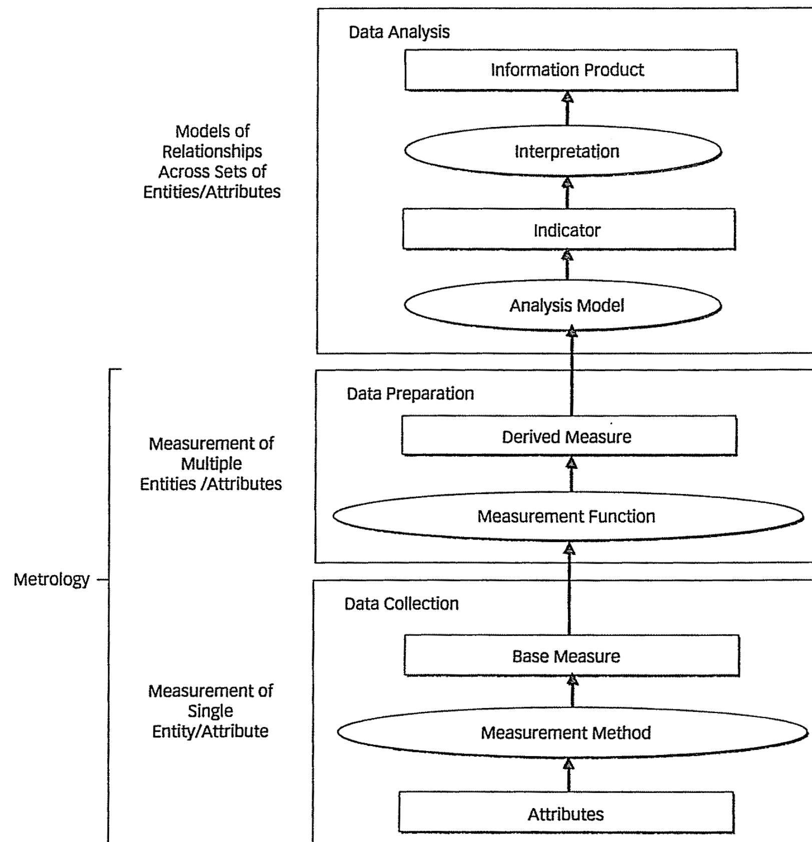

Figure 5.4: Process view of the ISO/IEC 15939 standard

The distinction between measurement and assessment in the quality assessment metamodel is compliant with the ISO/IEC 15939 standard [ISO/IEC, 2007a]. As shown in Figure 5.4, the measurement package focuses on concepts that are reminiscent of metrology (i.e., data collection and data preparation), whereas the assessment package addresses concepts that relates to the interpretation of measures aiming to satisfy an information need. The remainder of this section details each package of the quality assessment metamodel.

5.2.1 Project package

The **project package** provides constructs dedicated to the characterisation of the relevant *measurable entities* (e.g., diagrams, files, model transformations, etc.) present in the software project. The project component of any MoCQA model instantiates concepts of the project package in order to model a subset of the software project (i.e., a subset of the deliverables, processes and features that constitutes the project) that is investigated as part of the quality assessment process. The aim of this modelling effort is twofold. First, it aims to formalize the elements that have to be available in order to pursue a sound quality assessment (i.e., a quality assessment that satisfies the information needs of each involved stakeholder). Besides, it provides a proactive quality-related perspective on the software project (i.e., a view of the elements that require specific attention in order to lead to quality and a view on the elements that have to be refined in case of unsatisfying assessment results).

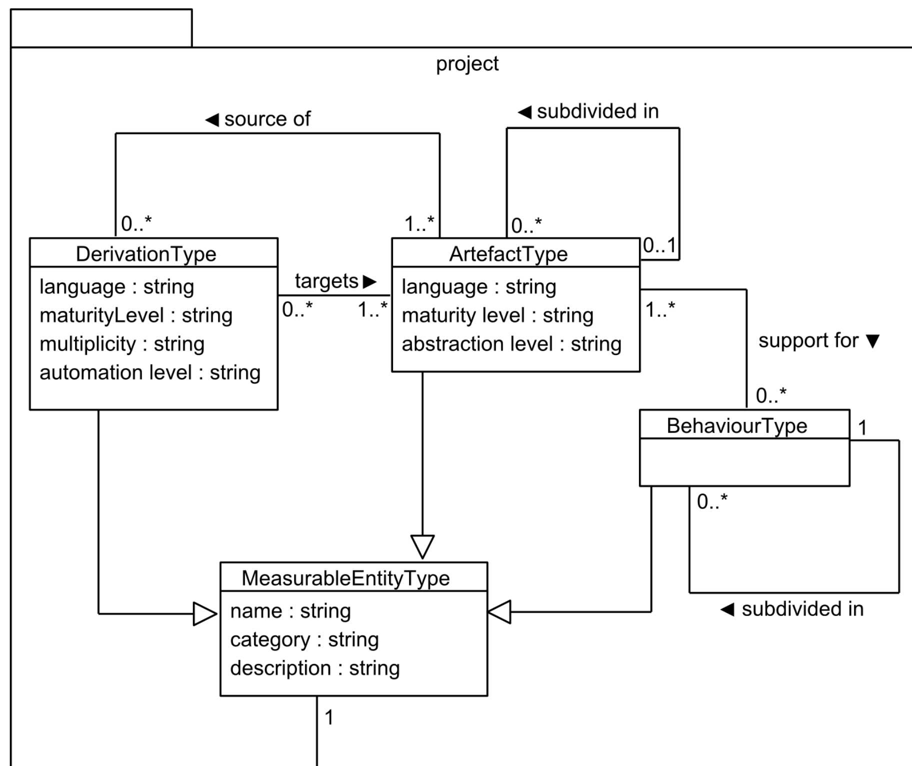

Figure 5.5: Project package of the quality assessment metamodel

Figure 5.5 provides a detailed view of the project package. As shown in the figure, the project package provides constructs aiming to model measurable entity *types*. This implies that each construct from this package defines an entire entity population. The more specific the information on an entity type is, the

smaller the entity population will be. For instance, if we define a measure for the measurable entity type *Java class* each Java class present in the project will be measured during the assessment step of the methodology. However, if we define a measure for the measurable entity type *Java class from package X*, this would reduce the number of instances taken into account during the assessment phase. Finally, if the defined measurable entity type is *Java class Y from package X*, only this specific class will be measured during the assessment step.

Additionally, we may define measurable entity types as collection of all entities of this type by adding the keyword "collection" in front of the name. This keyword specifies that the measurement or estimation methods associated to the entity type through base attributes will not be performed on each instance of the entity type (which is the default semantics for an association between a measurable entity type and an attribute) but on the collection of all existing entities of this type. This technique is provided in order to avoid unnecessary additional constructs. For instance, a "Java class" may be associated with a "size" attribute but the semantics of this association implies that for each Java class, the size of this specific class will be evaluated (e.g., based on the number of methods, attributes in the class). If one wants to rely on the number of Java classes for one function or assessment model, the size attribute should be associated with a "package X" artefact type or "source code" artefact type, since the number of classes contained in a package or in the code characterises the package (or the code) and not the class. In order to avoid the modelling of such containers when they are not required, the collection keyword may be used. This techniques remains coherent with the semantics of the entity types. Indeed, the strict interpretation of such an element is "all collection of all entities X are relevant in our quality assessment process". As for the example of the "Java class Y from package X", only one instance complies to the entity type declaration and therefore produces the intended result.

As shown in Figure 5.5, the project package provides three types of constructs: **artefact types**, **derivation types** and **behaviour types**. These concepts are all subclasses of a concept named **measurable entity type** which bridges the project package with the measurement package. The project package of the quality assessment metamodel therefore provides a generic typology of measurable entities. This typology results from an additional conceptualisation step performed on the project-level of the software quality ontology introduced in Chapter 3.

As shown in Figure 5.6, the entities may be divided into two categories: deliverable-related and process-related. These two types of entities are related to each other through transformational relationships (i.e., a process to transform one or more deliverables into new deliverables). In the process package, deliverable-related types of entity are encapsulated by the artefact type construct, while the derivation type constructs represent process-related types of entities The differ-

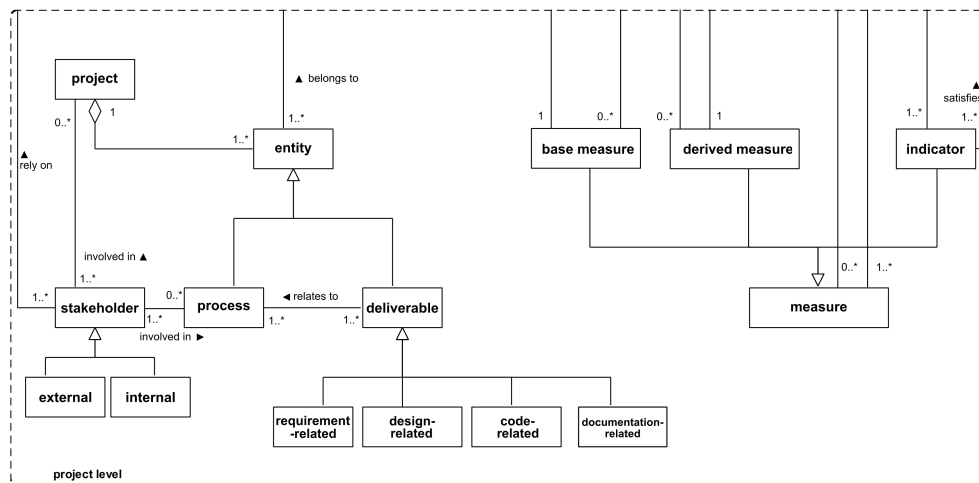

Figure 5.6: Project level of the software quality ontology

ence between deliverable/process and artefact/derivation will be explained in the remainder of this section. Figure 5.6 also shows that some deliverables may provide features (e.g., the source code) that are only measurable through external attributes. These features are encompassed in the behaviour type, since they provide a given behaviour at runtime.

Therefore, the project package allows the use of artefact types that may be associated with behaviour types they support and are interrelated through derivation types. Besides, the structure adopted by the project package remains compliant with our definition of software project:

> A collection of products (i.e., *artefacts*) linked together by transformational activities (i.e., *derivations*) and providing a collection of runtime features (i.e., *behaviours*) in order to satisfy a set of user's requirements.

The project package is therefore essential to implement the notion of explicit and integrated quality assessment modelling. Contrary to ISO/IEC standards, the project package and its constructs propose a more general and structured point of view than the usual software product viewpoint, which is the scope adopted by the ISO/IEC quality model. The software product in ISO/IEC standards is defined as a set of computer programs, procedures, and possibly associated documentation and data [ISO/IEC, 1999]. The project perspective adopted by the MoCQA models takes the same elements into account but provides a structured perspective on the software products. This structured perspective remains compliant with the notion of software ecosystem since an entire software system may be modelled as an artefact.

Artefact Types

Artefact types are the most straightforward project-related constructs allowed in a MoCQA model. These constructs allow the description of a relevant population of artefacts, which we define as follows:

Definition 5.3 (Artefact).
An identifiable item provided by the development team (in the large) or an external contributor that supports the overall software development process (and may be evaluated during the quality assessment process).

Note that although the term "artefact" is widely used in Software Engineering, it adopts a specific meaning in the context of this dissertation. The term **artefact** is a synonym for the concept of elementary artefact, defined as *a self sufficient piece of information comprised in a global artefact (i.e., specification, design or code) whose granularity is variable so that it is possible to define elementary artefacts with more or less important scopes* [Vanderose and Habra, 2008]. According to this definition, any resource that may be identified within a given software project and is therefore prone to measurement may be modelled in a MoCQA model through artefact type constructs. Artefacts may be regarded as a super-type for the deliverable concept defined in the CMMI framework . According to the definition provided above, artefacts may sometimes be assimilated to deliverables (e.g., a UML diagram, a Java package), they also encompass the notion of resources (e.g., a database, a software versioning system) and are not restricted to items developed specifically by the team (e.g., a software development kit, a software library, an entire software system).

Table 5.1 details the attributes available to characterise an artefact type construct. As pointed out before, an artefact type construct helps describe a collection of artefacts sharing the same properties (i.e., a class of artefacts) and worth investigating in the context of the subsequent quality assessment. Each instance of an artefact type included in a MoCQA model states that the software project should contain at least one occurrence of artefact demonstrating the properties defined by the instance of artefact type in order to allow the assessment and satisfaction of one or more quality issues. For instance, an artefact type named "use case" in a given MoCQA model implies that the quality assessment process requires the collection of all use cases available in the software project.

Due to the fact that the MoCQA framework takes transformational processes and the evolution of the software development into account, MoCQA models have to allow the expression of this temporal aspect. As a matter of fact, beside its intrinsic properties, an artefact may evolve according to two dimensions throughout the software development life-cycle: the abstraction level and the maturity level. The level of abstraction of an artefact type characterises the level of de-

Attribute	Description
Name	Provides the primary characterisation of an artefact type, which may be generic (e.g., sequence diagram, class, observer pattern, etc.) or more specific (e.g., sequence diagram *SEQ001*, method *getAccount())* in order to reduce the number of instances in the defined entity population that will actually be considered during the quality assessment cycle.
Category	Classifies instances of the artefact type according to their role in the software development life-cycle (e.g, requirement-related, design-related, code-related, document-related, test-related, etc.).
Description	Provides an additional characterisation of the artefact type in order to complement the information provided by the **name** (e.g., the sequence diagram related to the use case X).
Language	Provides information on the language used to express the instances of this artefact type (e.g., UML, C++, semi formal English language, etc.).
Maturity Level	Provides a way to characterise the level of maturity of the instances of this artefact type (e.g., “before refactoring”, “in production”, “version X”, etc.).
Abstraction Level	Provides a way to characterise the level of abstraction of the instances of this artefact type. (e.g., “high-level”, “class without attributes and methods”, etc.)

Table 5.1: Attributes characterising an artefact type construct

tail the artefact actually displays. For instance, a class diagram may contain only classes of a specific architecture, whereas another class diagram may provide more concrete information, such as the attributes of these classes, the types of the attributes, etc. The level of maturity provides information on both the “age” of the artefact and its level of completion. For instance, one might want to assess the source code with a given version number or at a certain stage of completion (e.g., “in production”, “draft”). These 2 attributes of artefact types also provide a way to distinguish artefacts that are used as input of a derivation from the output artefacts (e.g., a Java class before and after refactoring).

Artefacts type constructs may also be associated to other constructs of the quality assessment metamodel. Table 5.2 provides the list of authorised associations.

Attribute	Description
source of	Indicates that the instances of this artefact type are used as input of one or more **derivation types**.
support of	Indicates that the instances of this artefact type contribute to provide one or more **behaviour types**.
subdivided in	Indicates that the instances of this artefact type encompass one or more other **artefact types**.
characterised by	Indicates that the instances of this artefact possess one or more **attributes** of interest for the quality assessment process.

Table 5.2: Relationships involving artefact types

Illustration

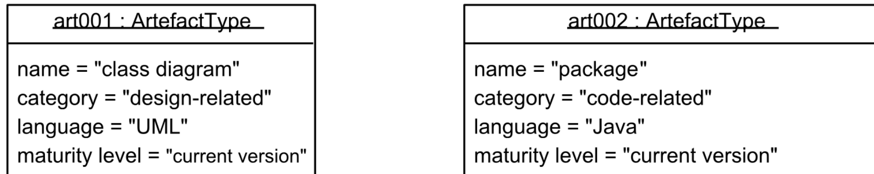

Figure 5.7: Two basic artefact types

Figure 5.7 illustrates the use of **artefact types** in a partial MoCQA model. This example, as well as further examples in this dissertation, rely on the UML object diagram notation as a concrete syntax for MoCQA models[2]. This notation is compliant with the abstract syntax defined by the quality assessment metamodel and therefore sufficient for illustrative purposes.

This example demonstrates a basic (and partial) MoCQA model stating that two types of entities are relevant and will be considered in the following quality assessment cycle: Java packages and class diagrams. As explained previously, the constructs of the project component of a MoCQA model represent classes of resources present in the software project. In this example, each existing class diagram and Java package is considered as a separate measurable entity that will be used as input in the investigation of a given quality issue.
Figure 5.8 elaborates on the first example in order to illustrate the "subdivided in" relationships. This new MoCQA model expresses the fact that each class diagram, each Java package and each Java class contained in each of these packages is a relevant measurable entity for the current quality assessment cycle. Since the

[2]Note that this notation is a variation of the UML object diagram since the links are oriented, in order to improve the legibility of the models

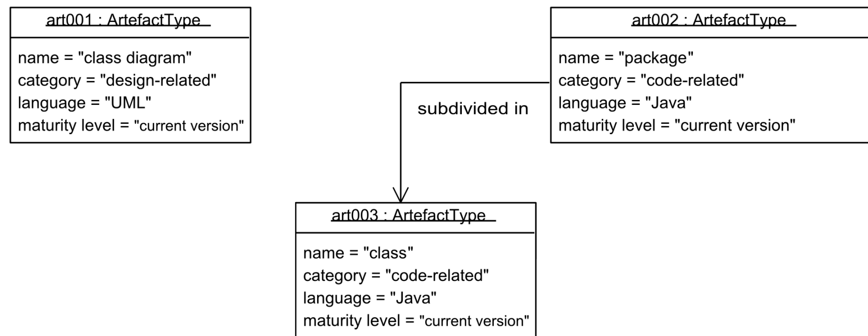

Figure 5.8: An artefact type with children artefact types

name of the artefact type provides more control on the size of the entity population, we may provide a more focused description of the measurable entities as shown, in Figure 5.9.

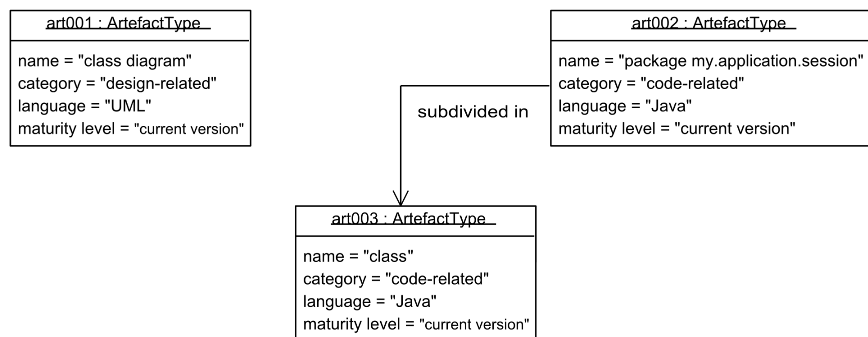

Figure 5.9: Artefact types with a reduced entity population

In that case, the scope of relevant measurable entities remains the collection of all class diagrams present in the software project but is restricted to the Java package `my.application.session` and each Java class contained in this specific package. In order to provide an even more focused scope, we may also provide a more specific name for the associated artefact type.

As shown in Figure 5.10, the scope of relevant code-related measurable entities in this last example is now restricted to the Java package `my.application.session` and *one* of its constitutive classes, that is, the Java class `login`. Depending on the way **attributes** are defined, the assessment of this project component could result in only one measure, applied to this specific Java class.

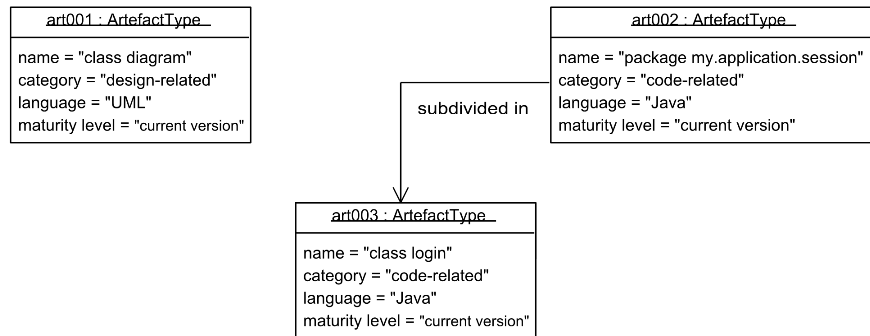

Figure 5.10: Artefact types with a very focused entity population

Behaviour Types

Behaviour types are constructs provided by the quality assessment metamodel to characterise types of runtime features that are evaluated during the quality assessment process. Behaviours are defined as follows:

Definition 5.4 (Behaviour).
An observable property provided by the software project at runtime and supported by executable artefacts in a given environment (and may be evaluated during the quality assessment process).

According to the SWEBOK guide [IEEE Computer Society, 2004]:

> At its most basic, a software requirement is a property which must be exhibited in order to solve some problem in the real world. [...] Hence, a software requirement is a property which must be exhibited by software developed or adapted to solve a particular problem.

Behaviours are therefore the counterpart of the requirements for a given software system. Ideally, for each requirement, the software system should demonstrate an appropriate set of behaviours. Additionally, [IEEE Computer Society, 2004] explains that an essential property of all software requirements is that they must be verifiable. As such, behaviours are relevant in the context of quality assessment.

In addition to their relation to requirements, behaviours share similarities with the notion of feature (i.e., "prominent or distinctive user-visible aspects, or characteristic of a software system [Kang et al., 1990]).

Table 5.3 details the attributes available to characterise a behaviour type construct. As for artefact types, a behaviour type construct helps describe a collection of behaviours sharing the same properties and worth investigating in the context of the subsequent quality assessment. Similarly, the preciseness of the name or description of the behaviour type influences the numbers of instances that

Attribute	Description
Name	Provides the primary characterisation of a Behaviour Type (e.g., display of user account information, crash of the application, loading screen, etc.).
Category	Classifies instances of the behaviour type according to their role regarding the runtime software system (i.e., if they are related to *functional* of *non functional* requirements/features).
Description	Provides an additional characterisation of the behaviour type in order to complement the information provided by the *name* (e.g., "behaviour related to use case X", behaviour provided during a certain amount of time).

Table 5.3: Attributes characterising a behaviour type construct

have to be considered. For instance, a behaviour type named `log-in` states that each occurrence of logging into the system is a measurable entity. A behaviour type named `screen freeze during log-in` would reduce the number of events that are considered, whereas a `first log-in` would describe a behaviour that happens only once.

Note that behaviours are measurable since they are observable phenomena but, contrary to artefacts, behaviours cannot be modified directly in order to improve the level of satisfaction of a quality issue. Any corrective action has to be taken on their supporting artefacts. In consequence, it is essential to provide this information in the MoCQA models to allow a better exploitation of the quality profile. Table 5.4 provides the list of authorised associations between behaviour types and others constructs.

Attribute	Description
support of	Indicates one or more **artefact types** the behaviour type is supported by.
subdivided in	Indicates that the instances of this **behaviour type** may be refined in one or more specifics behaviours (e.g., `calculation of mathematical results` and `calculation of average`)
characterised by	Indicates that the instances of this behaviour type possess one or more **attributes** of interest for the quality assessment process.

Table 5.4: Relationships involving behaviour types

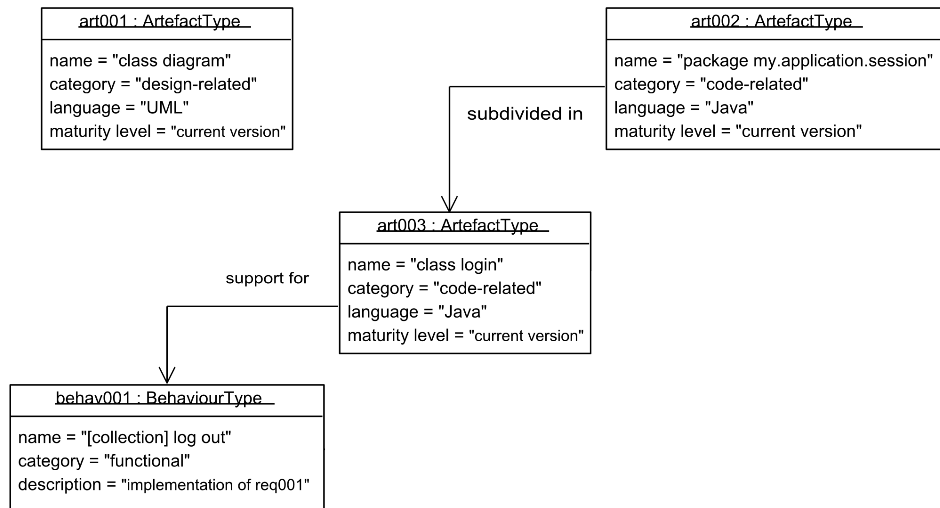

Figure 5.11: Example of a basic behaviour type

Illustration

Figure 5.11 illustrates our previous MoCQA model, completed with the characterisation of a behaviour type named `[Collection] log-out`. This behaviour type is associated with the `class 'login'` artefact type, is categorised as functional and is also informally linked to a specific user's requirement. As it is, the MoCQA model states that all logout action performed at runtime are considered relevant in the context of quality assessment process. However, the keyword "collection" specifies that we are not interested in evaluating each behaviour individually but all at once. This will have an impact on the kind of attribute that may be associated to this entity type.

Figure 5.12 illustrates another behaviour type, also related to the `login` class. This behaviour type is named `screen freeze at log out` and additionally characterised as *an unexpected unavailability of the log-out user interface*. This type of behaviour is *not* a desirable one (showing that errors may be modelled as well) and it is categorised as non-functional. It is also a subset of the previous behaviour type and is associated to the latter accordingly. This addition to the MoCQA model expresses that, in the context of our quality assessment process, all logout actions are relevant and specific bugs linked to these behaviours are considered as well. Contrary to the previous example, each occurrence of this behaviour will be measured during the assessment step.

Note that, due to fact that project-related constructs are defined as types of existing entities, some precautions must be observed when defining the relationships between these constructs. Some association could produce unexpected or misleading results. Figure 5.13 illustrates this case. This example of MoCQA

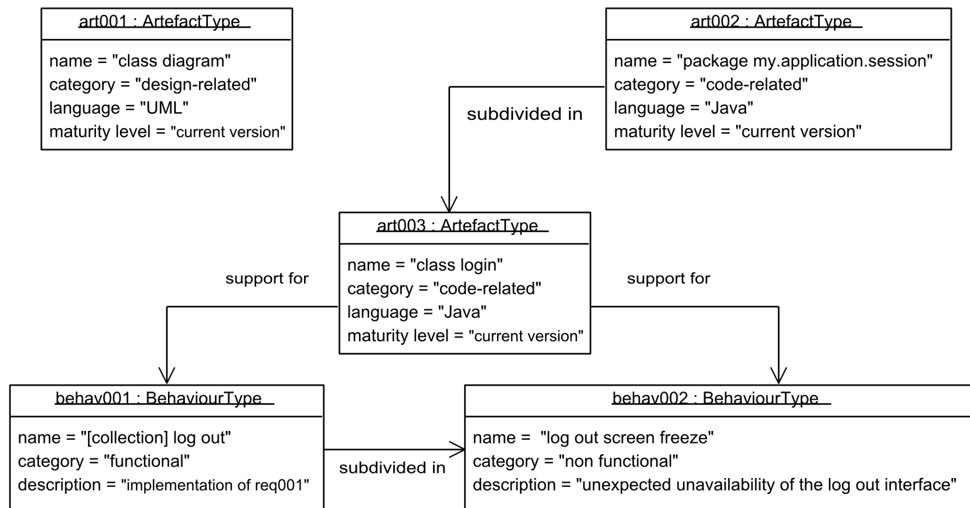

Figure 5.12: Example of more specific behaviour types

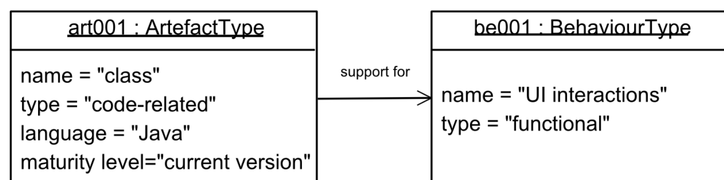

Figure 5.13: Misleading relationship between 2 measurable entity types

model intends to state that the quality assessment process is concerned by the collection of all possible interactions with the GUI and therefore associate the behaviour type with the wide-scoped class artefact type. However, due to the semantics of the constructs involved in this relation, the correct interpretation is : *for each Java class, the set of all GUI interactions supported by this class are relevant in the context of quality assessment.* This relation therefore provides multiples sets of behaviours but does not take into account the possible GUI-related functionalities provided by external libraries that are not written in Java. In that case, it would be better to associate the behaviour type with a coarse-grained "source code" artefact.

Derivation types

Derivation types are constructs provided by the quality assessment metamodel to characterise types of transformational activities occurring between artefacts. They may be evaluated during the quality assessment process. Derivations are defined as follows:

Definition 5.5 (Derivation).
A collection of more or less strictly defined principles describing a conversion relationship between one or more source artefacts and one or more artefacts derived from the first ones.

This concept shares similarities with the notion of model transformation. However, the term **derivation** is preferred to transformation in order to remain more general and avoid to restrict the scope of the concept. Indeed, model transformations, defined as the process of converting one model to another model of the same system in the MDA Guide [Miller and Mukerji, 2003], are a specific type of derivations. However, some derivations are not model transformation per se (e.g., documentation of the code).

Derivations may also be regarded as a super-type for the process concept defined in the CMMI framework. According to the definition provided above, derivations may sometimes be assimilated to processes (e.g., the implementation is the process/derivation that links UML packages to Java packages) but they also encompass the notion of automated process (e.g., Javadoc generation).

Attribute	Description
Name	Provides the primary characterisation of a derivation type (e.g., implementation, refactoring, documentation, etc.).
Category	Classifies instances of the derivation type according to their role regarding the transformation process they describe (i.e., endogenous/exogenous and horizontal/vertical).
Description	Provides an additional characterisation of the derivation type in order to complement the information provided by the **name** (e.g., derivations executed by team X, by tool Y).
Language	Provides information on the language used to express the instances of this derivation type, if it is formalised (e.g., QVT, ATL, etc.).
Multiplicity	Provides information on the number of input/outputs artefacts this derivation type uses (e.g., 1-to-many, many-to-many, etc.).
Maturity Level	Provides a way to characterise the level of maturity of the instances of this derivation type, if it is formalized (e.g., "version X", etc.).
Automation Level	Provides information on the tool-support for this derivation type (i.e., manual, semi-automated, fully automated).

Table 5.5: Attributes characterising a derivation type construct

Table 5.5 details the attributes available to characterise a derivation type construct. As for the other measurable entity types, a derivation type construct helps describe a collection of derivations sharing the same properties and worth investigating in the context of the subsequent quality assessment. However, in the case of derivation types, the entity population is mainly reduced through the artefact types it is associated to. For instance, a derivation type named `implementation` associated to an input artefact type named `class diagram` and an output artefact type named `Java class` states that any implementation activity is taken into account. Conversely, an input artefact type named `class diagram X` would reduce the derivations considered to any implementation activity using this specific class diagram. Modifying the output artefact type to `Java class Y` would leave only one relevant implementation activity.

A **derivation** is categorised according to a bidimensional characterisation (endogenous/exogenous, that is, relying on the same/a different metamodel, and horizontal/vertical, that is, conserving the same level of detail/adding details) inherited from the model transformation body of knowledge [Mens et al., 2005a]. A derivation is associated to a *language* that defines the model transformation language that has been use to express it or may be ignored if the derivation has not been formally defined.

Relationship	Description
source of	Indicates that the instances of this derivation type use one or more **artefact types** as input.
targets	Indicates that the instances of this derivation type provide one or more **artefact types** as output.
characterised by	Indicates that the instances of this derivation type possess one or more **attributes** of interest for the quality assessment process.

Table 5.6: Relationships involving derivation types

Due to their transformational nature, it may be useful to specify the number of input and output elements targeted by the derivation. Therefore, the derivation type may be characterised by a multiplicity that specifies the nature of the association with artefact types. Table 5.4 provides the list of authorised associations between derivation types and others constructs.

Illustration

Continuing our previous example, Figure 5.14 illustrates the addition of a derivation type named `implementation`. This derivation type serves the purpose of documenting the relationships between our two main artefact types of interest. It is categorised as exogenous since the source metamodel (i.e., UML metamodel) and the target metamodel (i.e., Java language metamodel) are different. It is also

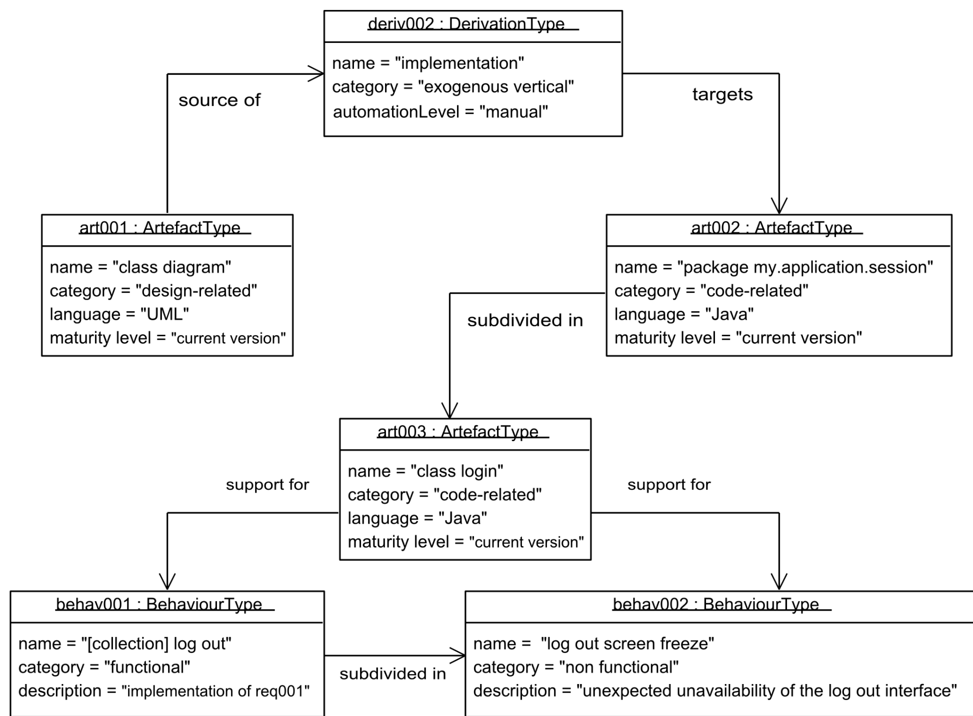

Figure 5.14: Example of derivation type

categorised as vertical since the implementation is supposed to add details to the architecture. The multiplicity is one-to-one since one class diagram is used to implement a Java package.

Note that as a MoCQA model is completed with more constructs, the global entity population is also reduced accordingly. In our case, the model may be interpreted as follows: for each class diagram in the software project, the implementation of this diagram that results in the output of a Java package named `my.application.session` is relevant to our quality assessment process. In consequence, only one class diagram satisfies these constraints and will be considered. Figure 5.15 shows the same example but in the case of a semi-automated transformation.

Related concepts of software engineering

As explained in Chapter 4, explicit quality assessment modelling implies the ability to express elements outside the scope of software quality. The remainder of this section reviews some project-level concepts of software engineering and how they translate in terms of MoCQA constructs. This list of concepts is not exhaustive but illustrates how the constructs of the quality assessment metamodel may be used to represent well-known concepts of Software Engineering.

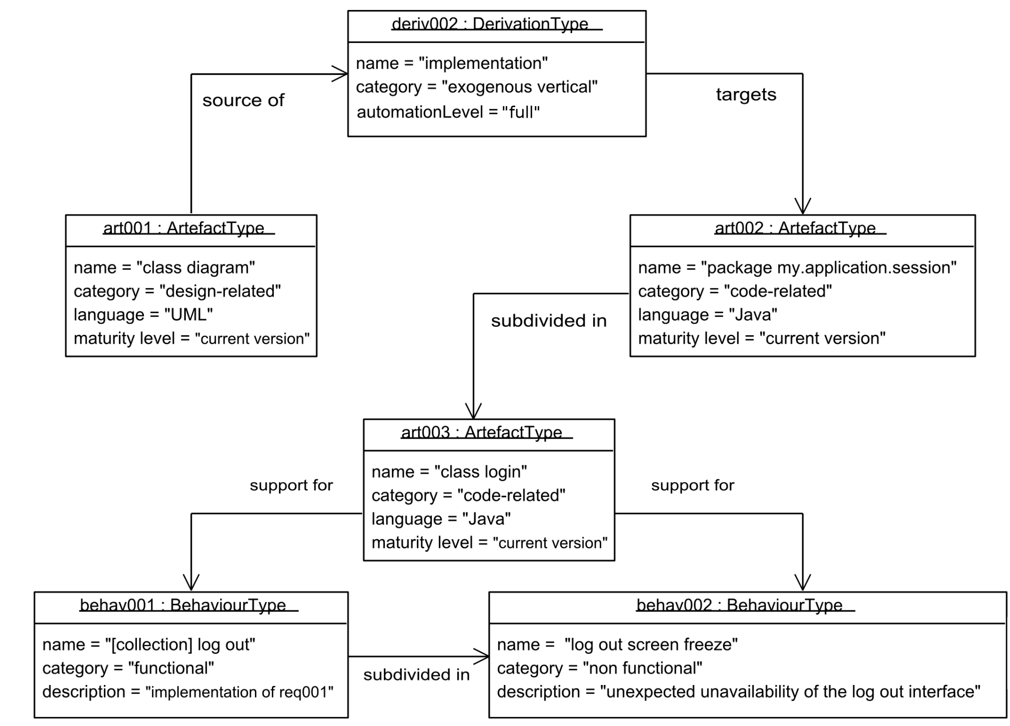

Figure 5.15: Example of automated derivation type

The notion of requirements may be expressed as *the set of all the existing (non overlapping) requirement-related* ***artefacts*** *within one* ***software project****.*

The notion of design (as a product) translates as *the set of all the existing (non overlapping) design-related* ***artefacts*** *within one* ***software project****.*

The source code may be expressed as *the set of all the existing (non overlapping) code-related* ***artefacts*** *within one* ***software project****.*

The implementation becomes *the set of all the existing (non overlapping)* ***derivations*** *between design-related* ***artefacts*** *and code-related* ***artefacts*** *within one* ***software project****.*

The design (as an activity) may be expressed as *the set of all the existing (non overlapping)* ***derivations*** *between requirement-related* ***artefacts*** *and design-related* ***artefacts*** *within one* ***software project****.*

Finally, the notion of refactoring may be expressed as *an endogenous* ***derivation*** *taking as input a code-related* ***artefact*** *and producing another whose semantics has been maintained unchanged while predefined syntax-related* ***quality indicators*** *have been improved.*

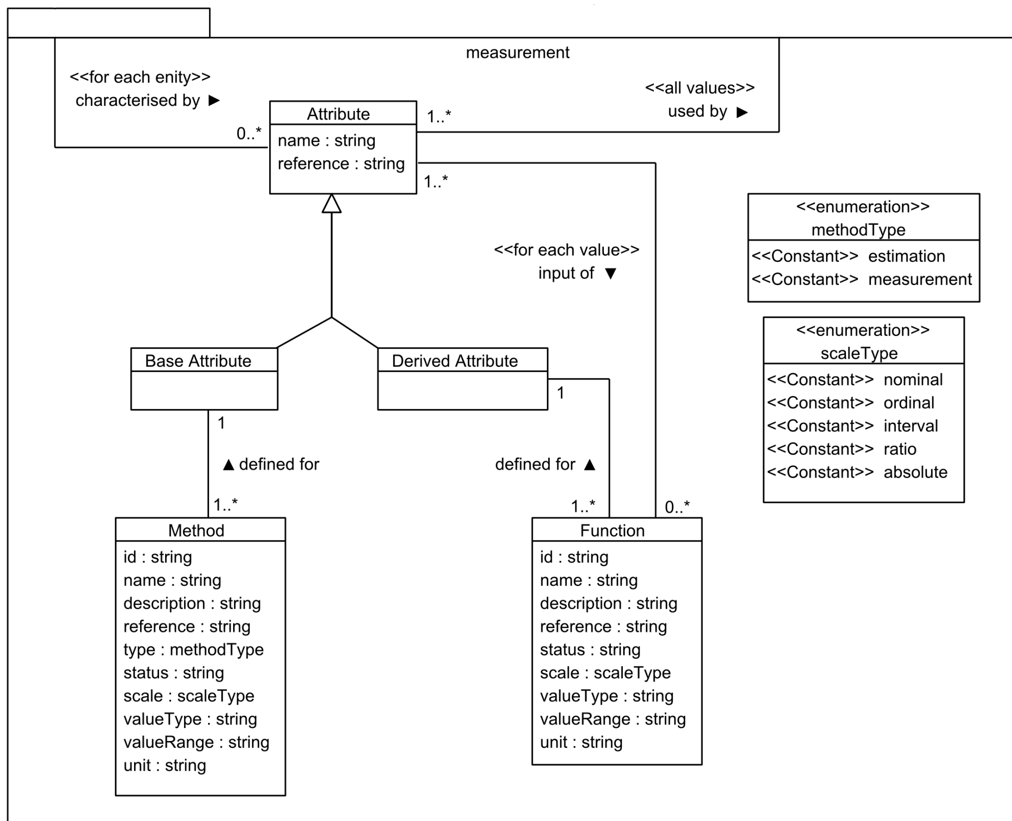

Figure 5.16: Measurement package of the quality assessment metamodel

5.2.2 Measurement package

The **measurement package** provides constructs dedicated to the definition of quantification methods (i.e., measurement or estimation methods) for the measurable entity types described in the project component of a MoCQA model.

Figure 5.16 provides a detailed view of the measurement package. As shown in the figure, the measurement package provides three main constructs: **attributes (base and derived)**, **method** and **function**. These elements are inherited from the measurement level of the software quality ontology introduced in Chapter 3.

The main difference between the measurement level of the ontology and the measurement package of the quality assessment metamodel is the strict interpretation of measure. The Software Measurement body of knowledge stresses the fact that quantification and measurement are not the same. Numbers may be assigned to a property, based on opinion or other heuristic methods. Numbers obtained that way do not possess metrologic properties and may provide erratic results. However, in the MoCQA framework, estimation methods may prove useful, especially at early stages of the development. Therefore, the measurement

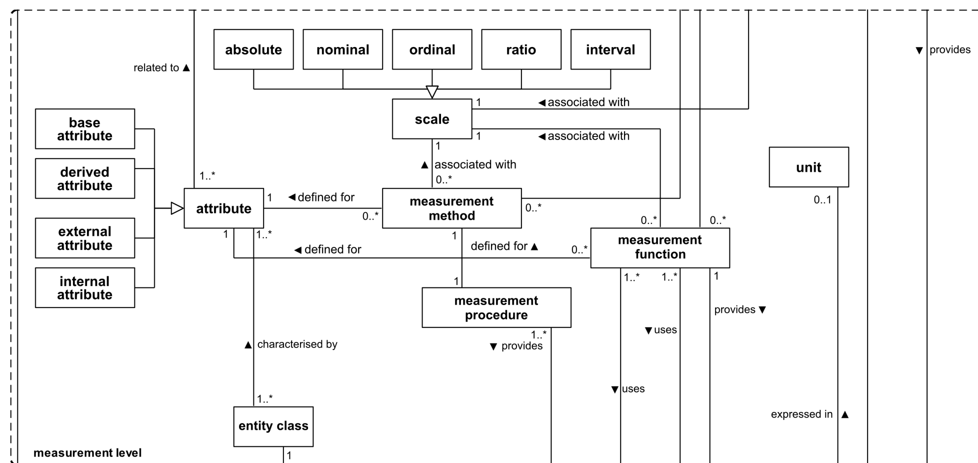

Figure 5.17: Measurement level of the software quality ontology

package provides a generic method construct that may be a measurement or an estimation method, according to the needs of the quality assurance team at a given moment of the quality assessment life-cycle. Yet, this construct has to be associated with an attribute, a scale and possibly a unit (concepts that are related to metrology) in order to structure the evaluation process and allow the evolution of the methods towards more accurate and reliable measurement methods as the software development life-cycle unfolds. For instance, a quality assurance team may want to use scoring cards in order to monitor the early stages of an architecture. The fact that this team had to associate its estimation method to a given attribute, a scale and a unit although it is not a measurement method, could help the quality assurance team switch more easily to a competent and validated measure as soon as it is applicable.

Another difference between the measurement level of the ontology and the measurement package is the absence of external/internal attributes. This absence is due to the way measurable entities are modelled. Indeed, the introduction of behaviour types allows the designer to ignore this distinction since an attribute linked to a behaviour type is by definition an external attribute, whereas an attribute associated to other types of entities is by definition an internal attribute.

Base Attributes

Base attributes are constructs provided by the quality assessment metamodel to characterise a property of a previously defined **measurable entity type** that will be evaluated during the quality assessment process. Base attribute constructs are a direct translation of the base attribute concept found in the measurement level of the software quality ontology.

Table 5.7 details the attributes available to characterise a base attribute con-

Attribute	Description
Name	Provides the primary characterisation of a **base attribute** that may originate from a referenced source (e.g., size, complexity, etc.) or be customised (e.g., occurrence, existence, etc.).
Reference	Provides information on the quality framework or measurement framework the base attribute originates from (e.g., ISO/IEC 9126, MOOD suite, etc.).

Table 5.7: Attributes characterising a base attribute construct

struct. In order to provide more information regarding the overall adequateness of the assessment process modelled, an optional reference may be specified in order to relate the base attribute to an identified framework or paper. In the case of a customised base attribute (i.e., an attribute with no reference), the name must reflect accurately the targeted property of the evaluated entity. As explained before, the distinction between internal and external attributes (i.e, attributes which can be measured purely in terms of the entity being measured or with respect to how the entity relates to its environment, respectively) made by several frameworks (including ISO/IEC 9126:2001) is not defined by the user but based on the nature of the measurable entity.

The **base attribute** may be associated to a **measurable entity type**. In comparison with the ontology, this association is different. As a matter of fact the quality assessment model does not provide base measure constructs. This design choice is induced by the iterative nature of our quality assessment methodology. In order to allow the evolution of the base measures used in our analysis models, the base attributes are used as a constant that allow to “plug” measurement or estimation methods into the model. While the evaluation method may change, the base attribute remains. Therefore, in the framework, a base measure may be defined as the value obtained through an evaluation method defined for a base attribute of a single entity. Note that due to the fact that entities are defined as type in the project package, a base attribute may result in one or more measurement/estimation values (i.e., one value for each instance of the entity type present in the software project). Table 5.8 provides the list and the semantics of authorised associations between base attributes and other constructs.

Illustration

Figure 5.18 illustrates how base attributes may be added to our example from Section 5.2.1. In this example, three base attributes are defined, one for behaviour type `behav001`, two for `behav002`. The basic idea of the MoCQA model at this stage is to count the number of logout actions performed and the number of

Relationship	Description
characterised by	Indicates the measurable entity type this **base attribute** characterises. The semantics of this association is: each instance of the entity type possesses the property defined by the base attribute.
defined for	Indicates the **method** that intends to evaluate this **base attribute**. The semantics of this association is: for each occurrence of this base attribute required during the quality assessment process, the method is used to produce a value assigned to the property.
input of (function)	Indicates that the measurement/estimation values of this **base attribute** are used as input of a **function** in order to measure/estimate the values of a derived attribute. The semantics of this association is: each value assigned to this attribute for each instance of the entity type is used as an input of the function to produce a distinct value.
input of (assessment model)	Indicates that the measurement/estimation values of this **base attribute** are used as input of **assessment model** in order to provide a **quality indicator**. The semantics of this association is: the set of all values assigned to this attributes is used as an input of the assessment model to produce a unique value.

Table 5.8: Relationships involving base attributes

screen freezes experienced in order to derive a ratio that will be exploited as an indicator. The notion of aggregated (or collection of) measurable entity types shows its relevance once base attribute are added to the MoCQA model.

Behaviour type `behav002` has been defined as a regular measurable entity type. Therefore, any base attribute associated to `behav002` will be evaluated for *each* occurrence of this behaviour type. In order to respect the semantics of the project package constructs (i.e., "for each instance of measurable entity type, let's evaluate a specific attribute"), we have to define a base attribute that is applicable to each occurrence of the behaviour type. In our example, the base attribute that allows us to count the total number of screen freeze and is

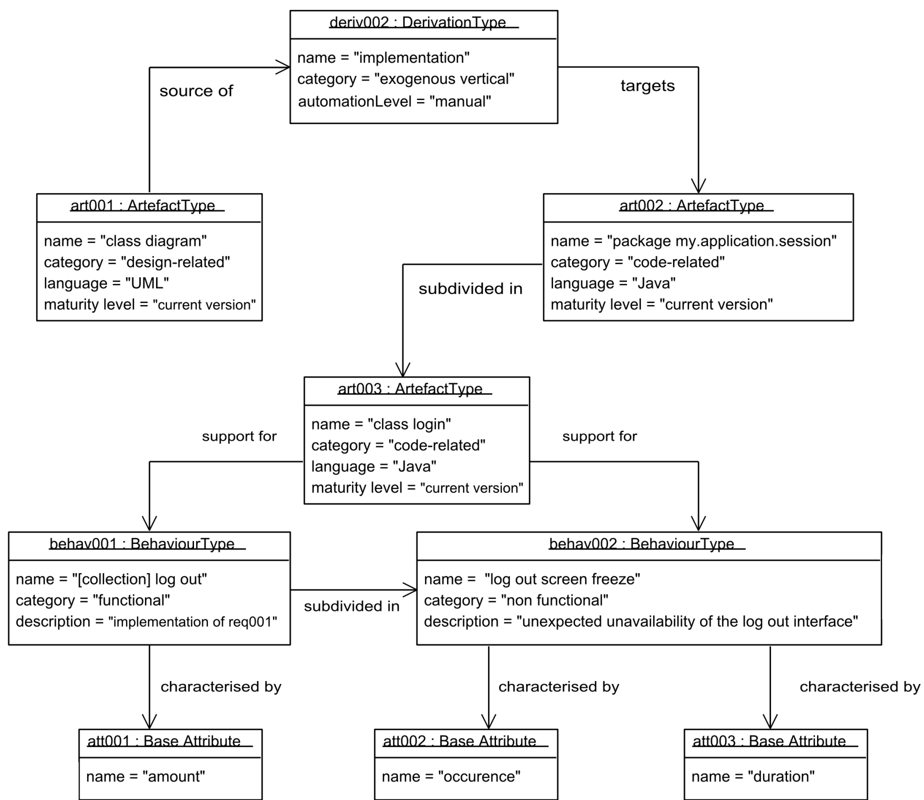

Figure 5.18: Example of base attributes

applicable to each separate occurrence of screen freeze is the `occurrence` base attribute. This base attribute states that for each instance of screen freeze, we want to evaluate its occurrence. Additionally, the base attribute `duration` is also associated to the behaviour type since it may be useful to discriminate critical freezes from regular and acceptable lag.

On the other hand, behaviour type `behav001` has been defined as an aggregated measurable entity type (i.e., a collection of all log out actions). As explained earlier, it may be relevant to consider a collection of behaviour types. This modelling convention permits the respect of the same semantics across artefacts and behaviours. In the strict semantic interpretation of MoCQA models, behaviour type `behav001` therefore states that each instance of all possible logout actions is taken into account. The base attribute `amount` (i.e., the amount of instances in the collection) may therefore be applied.

Methods

Methods are constructs provided by the quality assessment metamodel to characterise a measurement or estimation method defined to evaluate a specific base

attribute. As explained before, methods are a generalization of the measurement method concept present in the measurement level of the software quality ontology. As such, they share the same level of abstraction (i.e., a logical *sequence of operations*, *described generically*) and must be operationalised through measurement/estimation procedures in order to produce a value.

Attribute	Description
ID	Provides an identifier for the method, allowing to bind it to a procedure defined afterwards.
Name	Provides a name (if a name has been defined) to identify the method (e.g., McCabe's Cyclomatic Number, NLOC, etc.).
Description	Provides the logical sequence of operations to apply this measurement or estimation method.
Reference	Provides information on the quality framework or measurement framework the method originates from (e.g., ISO/IEC 9126, MOOD suite, etc.).
Type	Provides information on the stance of the method regarding software measurement (i.e., measurement or estimation).
Status	Provides information on the maturity of the method (i.e., experimental, theoretically validated, empirically validated or fully validated).
Scale	Provides the scale associated to each value produced by the method (i.e., nominal, ordinal, interval, ratio, absolute).
Value Type	Provides the type associated to each value produced by the method (e.g., integer, real, string, etc.).
Value Range	Provides an interval of values (of the same **value type**) that represents the lower and upper bounds for each value produced by the method.
Unit	Provides the unit associated to each value produced by the method (e.g., function points, line of codes, etc.).

Table 5.9: Attributes characterising a method construct

Relationship	Description
defined for	Indicates the **base attribute** this **method** intends to evaluate. The semantics of this association is: for each occurrence of the base attribute required during the quality assessment process, this method is used to produce a value assigned to the property.

Table 5.10: Relationships involving methods

Table 5.9 details the attributes available to characterise a method construct. The attributes may be divided in three categories. The first category of attributes (i.e., ID, name, description, reference) addresses the identification of the method, both internally to the measurement plan and regarding the software quality body of knowledge. The second category of attributes (i.e., category, status) provides information intended to increase the awareness of the quality assurance team regarding the robustness of their measurement plan. Finally, the last category of attributes (i.e., scale, value type, value range and unit) helps specify the properties of the values produced by the measurement or estimation method.

Method constructs are exclusively associated to based attribute constructs as shown in Table 5.10.

Illustration

As shown in Figure 5.19, each base attribute of our previous example is now associated with a suitable measurement or estimation method. Base attribute `att001` is associated with a counting method (i.e., the measurement consists in counting the number of log out actions performed during a time-frame still to specify). For instance, this method may translate as a procedure that relies on the log of the application to provide an actual number. As explained before, since there is only one occurrence of the measurable entity type, only one value will be produced by this method for this attribute.

Base attribute `att002` is associated to an estimation method that consists in reporting the occurrence of a screen freeze. For instance, this method may be translated into a procedure that relies on the log of a help desk to point towards the occurrences of a freeze. Base attribute `att003` is also associated to a measurement method that simply measure the duration of the screen freeze. This method may eventually rely on the log of the application as well. In consequence, each occurrence of screen freeze will be associated to a “flag” stating that it occurred and a duration in seconds.

Derived Attributes

Derived attributes are constructs provided by the quality assessment metamodel to characterise a property of a previously defined **measurable entity type** that will be computed on the basis of the values of other base or derived attributes. Derived attribute constructs are a direct translation of the derived attribute concept found in the measurement level of the software quality ontology.

Table 5.11 details the attributes available to characterise a base attribute construct. Basically, derived attribute constructs are similar to base attribute constructs, except for the way they are evaluated. Table 5.12 provides the list and the semantics of authorised associations between derived attributes and other constructs.

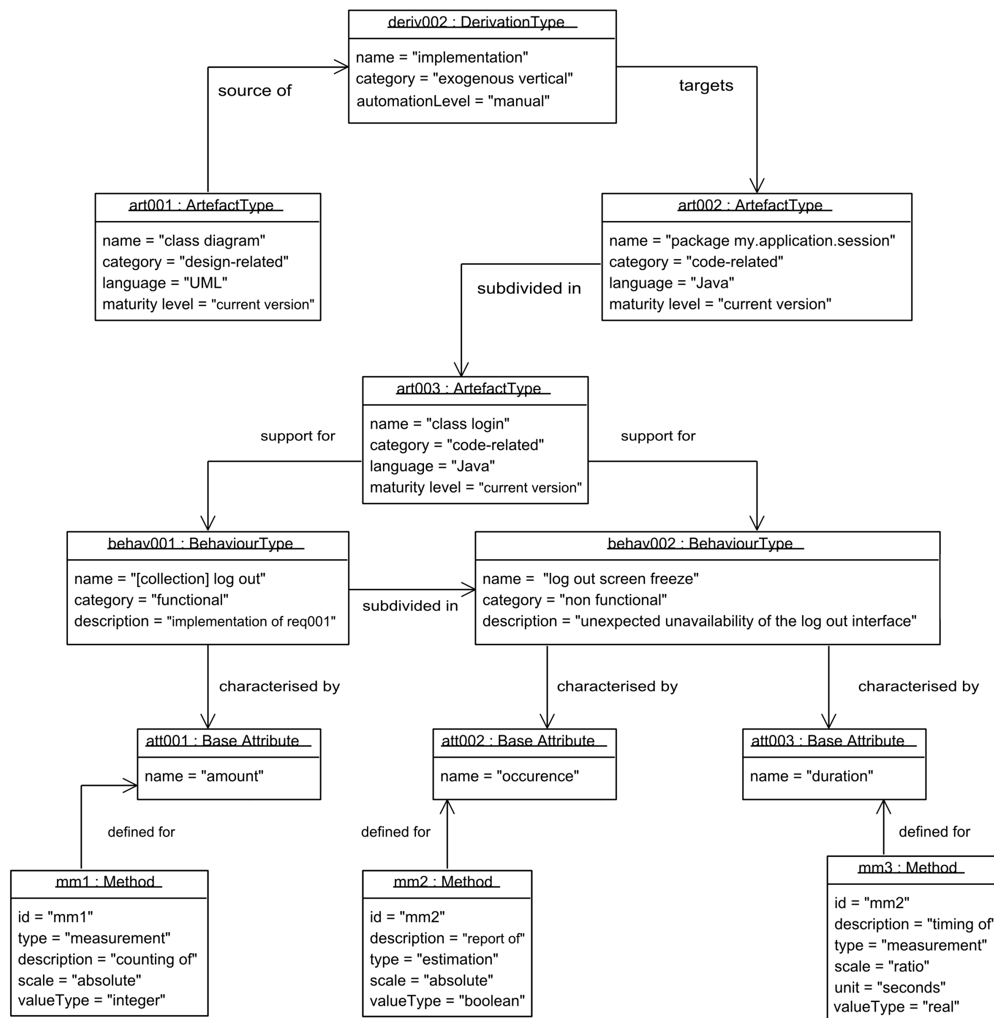

Figure 5.19: Example of measurement/estimation methods

Attribute	Description
Name	Provides the primary characterisation of a derived attribute that may originate from a referenced source (e.g., size, complexity, etc.) or be customised (e.g., occurrence, existence, etc.).
Reference	Provides information on the quality framework or measurement framework the base attribute originates from.(e.g., ISO/IEC 9126, MOOD suite, etc.).

Table 5.11: Attributes characterising a derived attribute construct

Functions

Functions are constructs provided by the quality assessment metamodel to characterise an algorithm or calculation defined to evaluate a specific derived at-

Relationship	Description
characterised by	Indicates the measurable entity type this **derived attribute** characterises. The semantics of this association is: each instance of the entity type possesses the property defined by the derived attribute.
input of (function)	Indicates that the measurement/estimation value of this **derived attribute** is used as input of a **function** in order to measure/estimate the values of a derived attribute.
input of (assessment model)	Indicates that *all* measurement/estimation value of this **derived attribute** is part of the input of the **assessment model** in order to provide a **quality indicator**.

Table 5.12: Relationships involving derived attributes

tribute. Function constructs are direct translations of the measurement function concept found in the measurement level of the software quality ontology.

Table 5.13 details the attributes available to characterise a function construct. Fundamentally, function constructs are similar to method constructs, except for the nature of the description (functions are exclusively calculations and algorithms) and the association that are authorised, shown in Table 5.14.

Illustration

As shown in Figure 5.20, a derived attribute `criticality` has been associated to behaviour type `behav002`. Once again, it means that for each occurrence of a screen freeze during the log out action, a value will be given to this attribute. In order to compute the criticality of the freeze, a simple function is associated to the derived attribute. This function consists in associating a value of 1 if the screen freeze had a duration of more than 2 seconds and a value of 0 in other cases. The `criticality` attribute will therefore be represented by an array of 0/1 values that may be used by an assessment model.

5.2.3 Assessment package

The **assessment package** provides constructs dedicated to the definition of a structure of quality goals. It also manages the description of how their related indicators rely on the constructs of the project and measurement components of a MoCQA model.

Attribute	Description
ID	Provides an identifier for the function, allowing to bind it to a concrete algorithm defined afterwards.
Name	Provides a name (if a name has been defined) to identify the function.
Description	Provides the algorithm or calculation performed by the function in general terms (same level of abstraction than method).
Reference	Provides information on the quality framework or measurement framework the function originates from.(e.g., ISO/IEC 9126, MOOD suite, etc.).
Status	Provides information on the maturity of the function (i.e., experimental, theoretically validated, empirically validated or fully validated).
Scale	Provides the scale associated to each value produced by the function (i.e., nominal, ordinal, interval, ratio, absolute).
Value Type	Provides the type associated to each value produced by the function (e.g., integer, real, string, etc.).
Value Range	Provides an interval of values (of the same **value type**) that represents the lower and upper bounds for each value produced by the function.
Unit	Provides the unit associated to each value produced by the function (e.g., function points, line of codes, etc.).

Table 5.13: Attributes characterising a function construct

Relationship	Description
defined for	Indicates the **derived attribute** this **function** intends to evaluate.
input of	Indicates the measurement/estimation value of which **attribute** are used as input of this **function** in order to measure/estimate the value of a derived attribute.

Table 5.14: Relationships involving functions

Figure 5.21 provides a detailed view of the assessment package. As shown in the figure, the assessment package provides three constructs: **quality issues**, **assessment models**, **quality indicator** and **interpretation rule**. These elements are adapted from the assessment level (Figure 5.22) of the software quality ontology introduced in Chapter 3.

The main difference between the assessment-level of the ontology and the assessment package of the quality assessment metamodel is that three concepts of the former are integrated into one in the latter. As a matter of fact, the

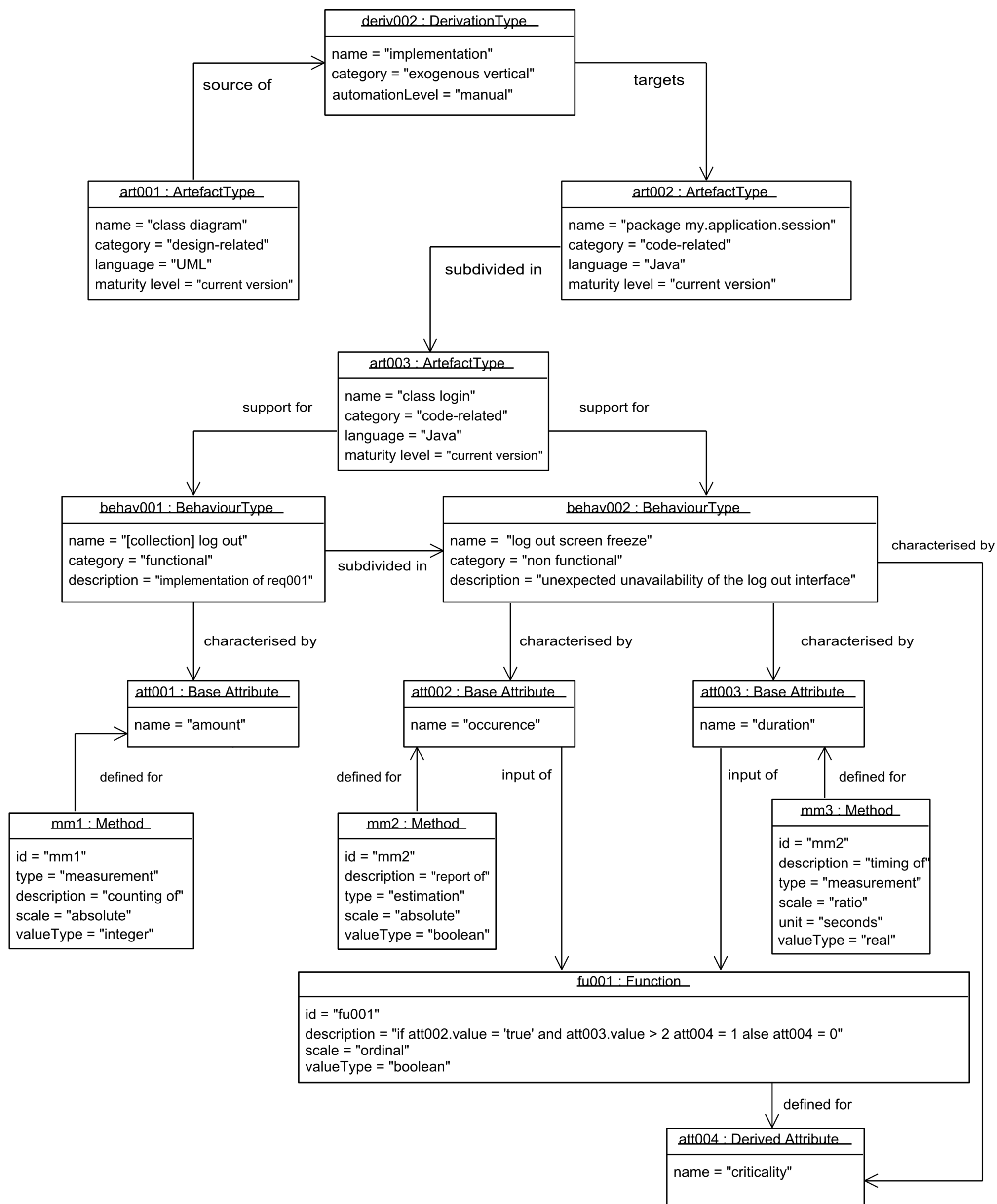

Figure 5.20: Example of derived attributes and functions

quality issue construct incorporates the concepts of quality factor, information need and reference to a quality model. The assessment model and interpretation rule constructs are directly inherited from the analysis model and decision criteria concepts (respectively). Finally, the quality indicator construct allows the characterisation of an indicator (defined in the project level of the ontology).

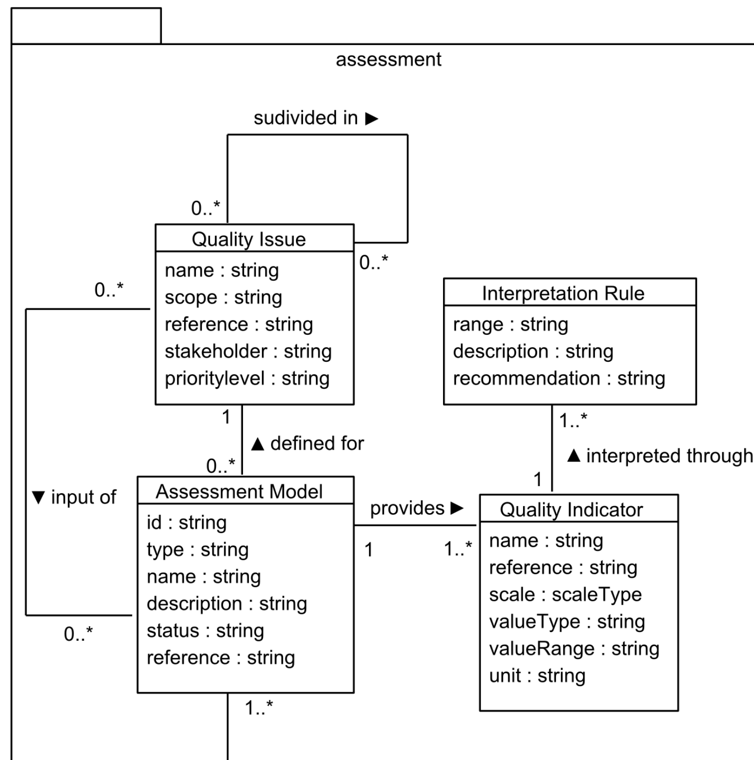

Figure 5.21: Assessment package of the quality assessment metamodel

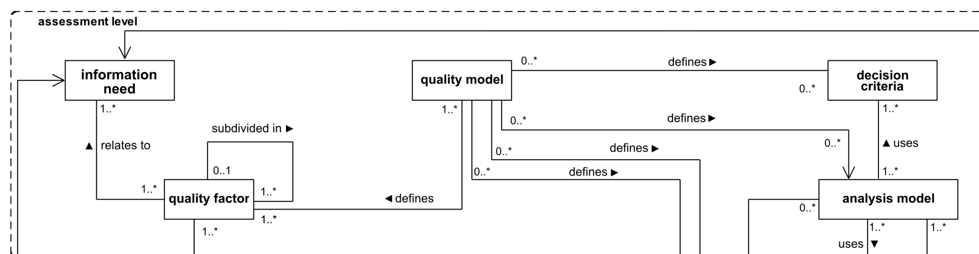

Figure 5.22: Assessment-level of the software quality ontology

Quality Issues

Fundamentally, **quality issues** are constructs provided by the quality assessment metamodel to characterise quality goals for the software project. In addition, quality issues encapsulate several concepts present in the assessment level of the software quality ontology. Concretely, quality issue constructs have a similar role to corporate objectives and tactical/measurement goals of the GQM/MEDEA approach even if they are *expressed* through quality factors (that may or may not originate from a specified quality model). In addition, they encapsulate an information need (as defined in ISO/IEC 15939) and are therefore associated to

specific stakeholders. They may be structured as a hierarchy and prioritized according to organisational needs. Finally, since quality assessment models define the quality-related aspects transversally to the entire software project, each quality issue construct has to be associated with a specific subset of the project for which it is relevant.

Attribute	Description
Name	Provides the quality factor targeted by the quality issue which may originate from a referenced source (e.g., maintainability, robustness, etc.) or be customised (e.g., time-to-market, cost-effectivness, etc.).
Reference	Provides information on the quality framework the quality factor originates from (e.g., ISO/IEC 9126, MOOD suite, etc.), if applicable.
Scope	Provides information on the subset of the software project in which this quality issue is relevant (i.e., application, code, design, a specific package, etc.).
Stakeholder	Provides information on the stakeholders this quality issue is defined for (i.e., management, customer, team X, etc.).
Priority Level	Provides information on the criticality of this quality issue according to the associated stakeholders.

Table 5.15: Attributes characterising a quality issue construct

Table 5.15 details the attributes available to characterise a quality issue construct. As explained before, due to the fact that MoCQA models intends to be operational, quality issues are basically quality goals. However, their primary characterisation is accomplished by way of a quality factor. Therefore, the reference attribute allows the specification of the source from which the quality factor originates. The scope attribute provides a way to define the specific part of the software project which is aimed at by this goal. This attribute also helps define the level of granularity of the goal, since the scope may be broad (e.g., the whole application) or very specific (e.g., a specific package or a class). The more specific the scope is, the more the MoCQA model is brought closer to an information product (as defined by ISO/IEC 15939). Indeed, if the MoCQA model defines a single quality issue whose scope is the same as a unique measurable entity type defined in the project component, the MoCQA model maps perfectly the Measurement Information Model. The stakeholder attribute and the priority level attribute manage the information need aspect and provide respectively a way to define who is interested in this goal and how critical it is to the stakeholder. Note that the priority level is voluntarily left free-form so that it can be used in any context (see Chapter 12 for an example of prioritized quality issues).

Quality issue constructs may be organised as a hierarchy. This structure may

be as simple as a given number of independent quality issues with no depth level (i.e., a 'flat' structure). In most cases, however, a precise hierarchy will be provided. The quality assessment metamodel offers two different ways to describe this hierarchical relation: the aggregation relationship or the composition relationship. The first one describes a quality issue with two or more sub-factors linked through an assessment model, which means that the children are used as input of an assessment model that will produce one or more quality indicators for the quality issue itself. The second one only describes a structure between the quality issue and its children, hence defining the parent issue as a multidimensional quality issue. Table 5.16 provides the list and the semantics of authorised associations between quality issues and other constructs.

Relationship	Description
defined for	Indicates the **assessment model** that intends to evaluate this **quality issue**.
input of (assessment model)	Indicates that the indicators of this **quality issue** are used as input of an **assessment model** in order to provide a **quality indicator**. Therefore indicates that this quality issue is part of an aggregation relationship.
subdivided in	Indicates that this quality issue is decomposed in one or more children **quality issues**. Therefore indicates that this quality issue is part of a composition relationship.

Table 5.16: Relationships involving quality issues

Assessment Models

Assessment models are constructs provided by the quality assessment metamodel to characterise an algorithm of calculation defined to produce one or more quality indicators assessing the level of satisfaction of a quality issue, based on attributes defined in the measurement component. Assessment models are a translation of the analysis model concept found in the assessment level of the software quality ontology.

Table 5.17 details the attributes available to characterise an assessment model construct. As we may see, assessment models share similarities with functions and therefore the two constructs have many attributes in common. The type attribute, however, is specific to the assessment model construct. This attribute is not to be confused with the type attribute of method constructs. The type of an assessment model provides a way to express the intent of the model. For

Attribute	Description
ID	Provides an identifier for the assessment model, allowing to bind it to a concrete algorithm defined afterwards.
Name	Provides a name for the assessment model, if applicable.
Description	Provides the algorithm or calculation performed by the assessment model.
Reference	Provides information on the quality framework or measurement framework the method originates from (e.g., ISO/IEC 9126, MOOD suite, etc.), if applicable.
Type	Provides information on the specific intent of the assessment model (i.e., estimation, prediction, etc.).
Status	Provides information on the maturity of the assessment model (i.e., experimental, theoretically validated, empirically validated or fully validated).

Table 5.17: Attributes characterising an assessment model construct

instance, one may use an attribute of a class diagram (e.g., size) to provide an indicator for the maintainability of the diagram. In that case, the assessment model would be a prediction model. Besides, the assessment model construct does not provide a characterisation of an output value. This is due to the fact that, contrary to a function, an assessment model may provide several *different* output values (i.e., quality indicators).

The description of the assessment model defines the relationship between the output quality indicator and the input attributes or quality issues (i.e., what is the quantitative impact of each attribute or child issue in the computation of the resulting quality indicator). These rules can take the form of an algebraic formula or a algorithm made of 'if-then' statements.

Note that each input of an assessment model (attribute or issue) is in fact an array of values. As for functions, the aim of an assessment model is to transform input values into output values, through a calculation process. However, the assessment model works on a different scale. Indeed, we have seen that an attribute is evaluated for each instance of its related entity type. The assessment model thus bridges the gap between the entity populations that have been measured and the scope of its related quality issue. For instance, if the scope is broad (e.g., the entire source code), the assessment model must provide rules that determine how many instances of the defined attributes and entities (e.g., structural complexity of a procedure) are to be taken into account (e.g., one of them, 60 percent or all of them). This allows the creation of a quality assessment that relies on a probing into the actual entities. It should be noted that assessment models are a crucial part of quality assessment modelling. They represent the link that binds mere measurement with actual meaningful quality assessment. Assessment models are

also the pivotal mechanism for the tailoring and fine tuning of a quality model within a given environment since they allow us to take environment factors in consideration. Consequently, the definition and validation (or at least the documentation of the rationale behind the model) of any assessment model should be cautiously taken care of.

Table 5.18 provides the list and the semantics of authorised associations between assessment models and other constructs.

Relationship	Description
defined for	Indicates the **quality issue** this **assessment model** intends to assess.
input of (quality issue)	Indicates that the indicators of the **quality issue** are used as input of this **assessment model** in order to provide a **quality indicator**.
input of (attribute)	Indicates that the values of the **attribute** are used as input of this **assessment model** in order to provide a **quality indicator**.
provides	Indicates one or more quality indicators computed through the **assessment model**.

Table 5.18: Relationships involving assessment models

Illustration

Figure 5.23 shows that one quality issue has been defined on top of our previous example. This element provides the fundamental rationale behind the measurement process described so far. The quality factor encompassed by the quality issue is the reliability of the `my.application.session` package (which is specified as the scope of the quality issue). Additionally, the quality issue specifies that this quality requirement has been expressed by the customer, which is the stakeholder for the quality issue.

As explained before, the quality issue expresses both a goal and an information need. As such, it requires an assessment model to link the quality issue with the selected attributes and their values. Assessment model `amod001` is defined for the previous quality issue. It uses base attributes `att001` and `att004` to provide the required assessment. The description of `amod001` states that the value of the output indicator will be computed on the basis of the ratio between the sum of the values contained in the `att004` array and the unique value associated to `att001`. If this ratio amounts to less than 0.2, the quality indicator would be assigned a 'OK' value. In other cases, the value assigned is 'KO'. Additionally, we may express the fact that this model is completely experimental and is therefore an

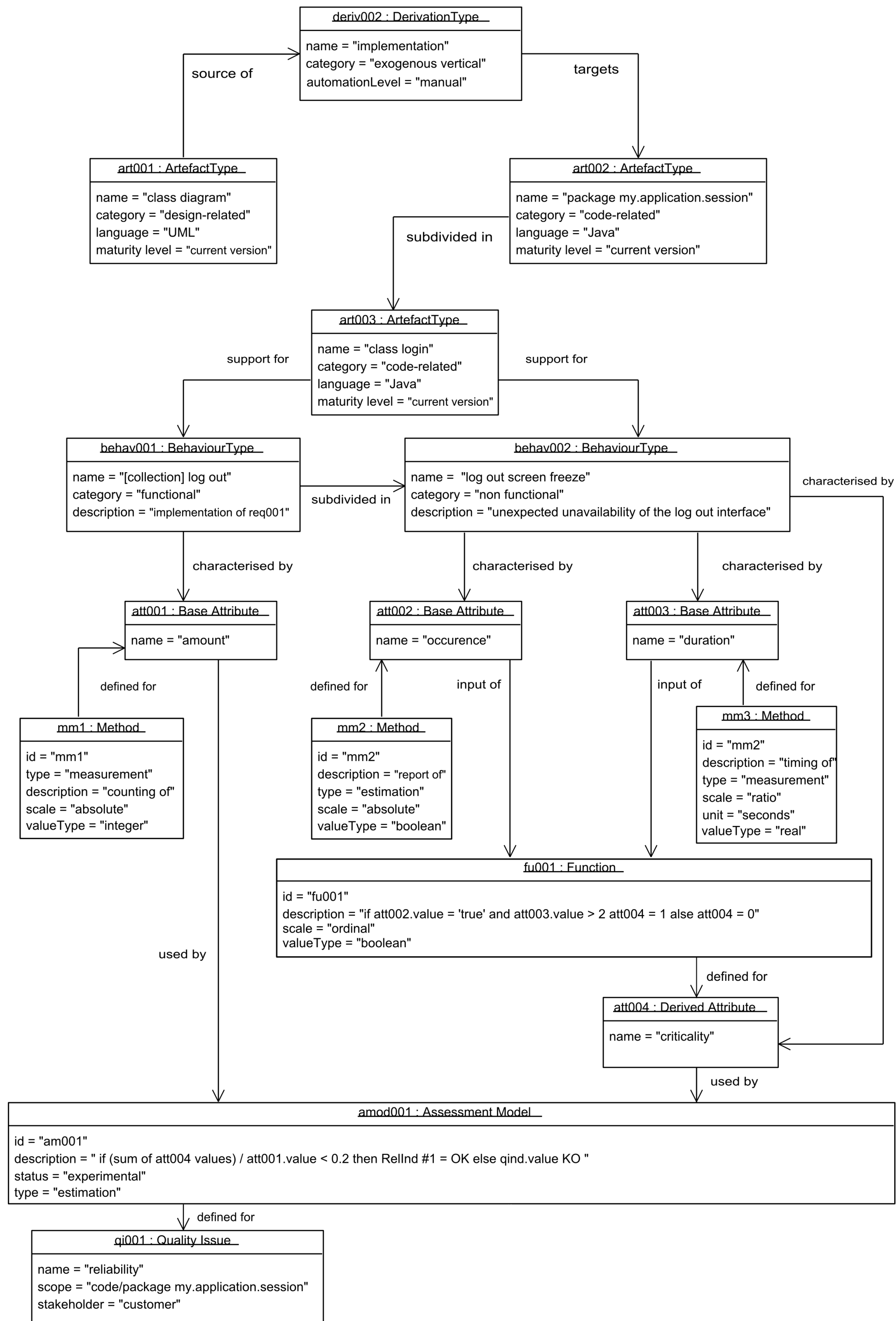

Figure 5.23: Example of quality issues and assessment models

attempt to answer to the information need that may be refined in the remainder of the quality assessment life-cycle.

Quality Indicators

Quality indicators are constructs provided by the quality assessment metamodel to characterise a specific measure or estimation produced by an assessment model in order to *interpret* the level of satisfaction of a quality issue. Quality indicators are an abstraction of the indicator concept found in the project level of the software quality ontology.

Attribute	Description
Name	Provides a name for the quality indicator, if applicable.
Reference	Provides information on the quality framework or measurement framework the quality indicator originates from (e.g., ISO/IEC 9126, MOOD suite, etc.), if applicable.
Scale	Provides the scale associated to the value produced by the quality indicator (i.e., nominal, ordinal, interval, ratio, absolute).
Value Type	Provides the type associated to the value produced by the quality indicator (e.g., integer, real, string, etc.).
Value Range	Provides an interval of values (of the same **value type**) that represents the lower and upper bounds for the value produced by the quality indicator.
Unit	Provides the unit associated to the value produced by the quality indicator (e.g., function points, line of codes, etc.), if applicable.

Table 5.19: Attributes characterising a quality indicator construct

Table 5.19 details the attributes available to characterise a quality indicator construct. In addition to the attributes designed to identify a quality indicator, the construct provides the same attributes designed to characterise a value as the function or method constructs do. As a consequence, a quality indicator may be numerical or not, is comprised into a predefined interval of relevant values associated with a scale and can be aggregated from several measurement values or even have the exact same value than one. The main difference between a measurement value and a quality indicator is that whereas the value is neutral and unattached to a meaning per se, the quality indicator is produced to be interpreted as a meaningful information regarding the achievement of a quality goal. As a matter of fact, a valid quality indicator has to be associated with a collection of interpretation rules that help give a meaning to the indicator, otherwise, the indicator is just useless. For instance, the functional size of a

given program is a measurement value. If we add simple rules of interpretation to define whether a project is small enough for a team with limited resource to cope with that is based on the functional size value, size becomes an indicator of the a quality issue that we may call "Feasibility". The quality indicator construct is compliant with the concept of indicator defined in ISO/IEC 15939 as *a measure providing an estimate or evaluation of specified attributes derived from a model with respect to defined information needs.*

As shown in Table 5.20, quality indicators are associated with a series of **interpretation rules** that allow the indicator to be more than just a neutral value and separates the quality indicator from the measure.

Relationship	Description
interpreted through	Provides one or more interpretation rules for the **quality indicator**.
provides	Indicates the **assessment model** this **quality indicator** is computed through.

Table 5.20: Relationships involving quality indicators constructs

Illustration

Figure 5.24 shows our previous example with a suitable quality indicator defined. This quality indicator is given a name for easier further reference (RelInd #1). It is logically assigned a binary value type (OK, KO) in order to be compatible with `amod001` and is associated with a nominal scale.

Interpretation Rule

Interpretation rules are constructs provided by the quality assessment meta-model to attach a defined meaning to a range of values the quality indicator may be comprised in. Interpretation rules are a translation of the decision criteria concept found in the assessment level of the software quality ontology.

Table 5.21 details the attributes available to characterise a quality indicator construct. Each interpretation rule is therefore a statement that helps bind the value of the quality indicator to a meaning regarding the quality issue it is defined for. Each interpretation rule may be regarded as a decision criterion as defined the software measurement terminology (see Section 3.1). Table 5.22 provides the authorised association between interpretation rules and other constructs, as well as its semantics.

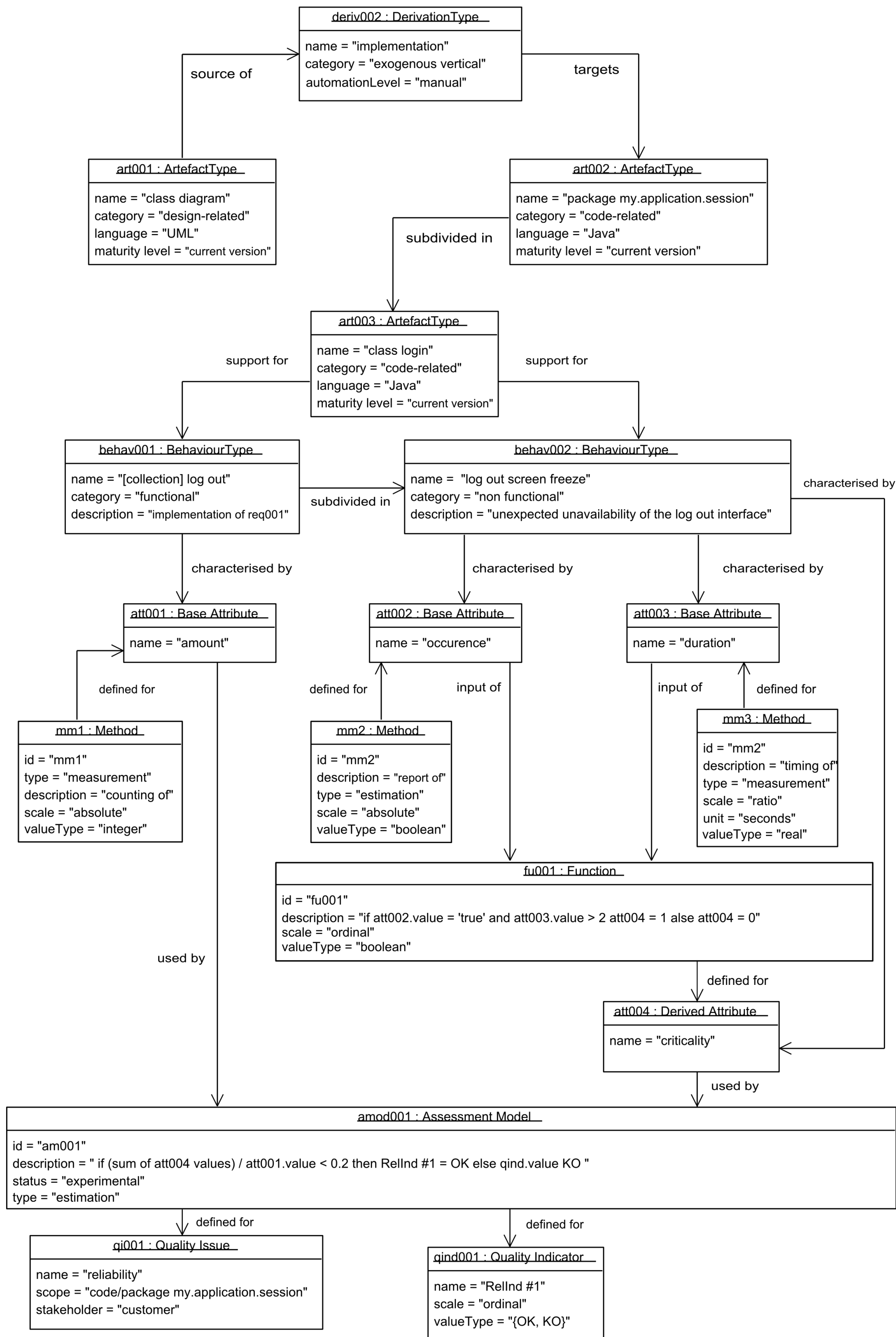

Figure 5.24: Example of quality indicators

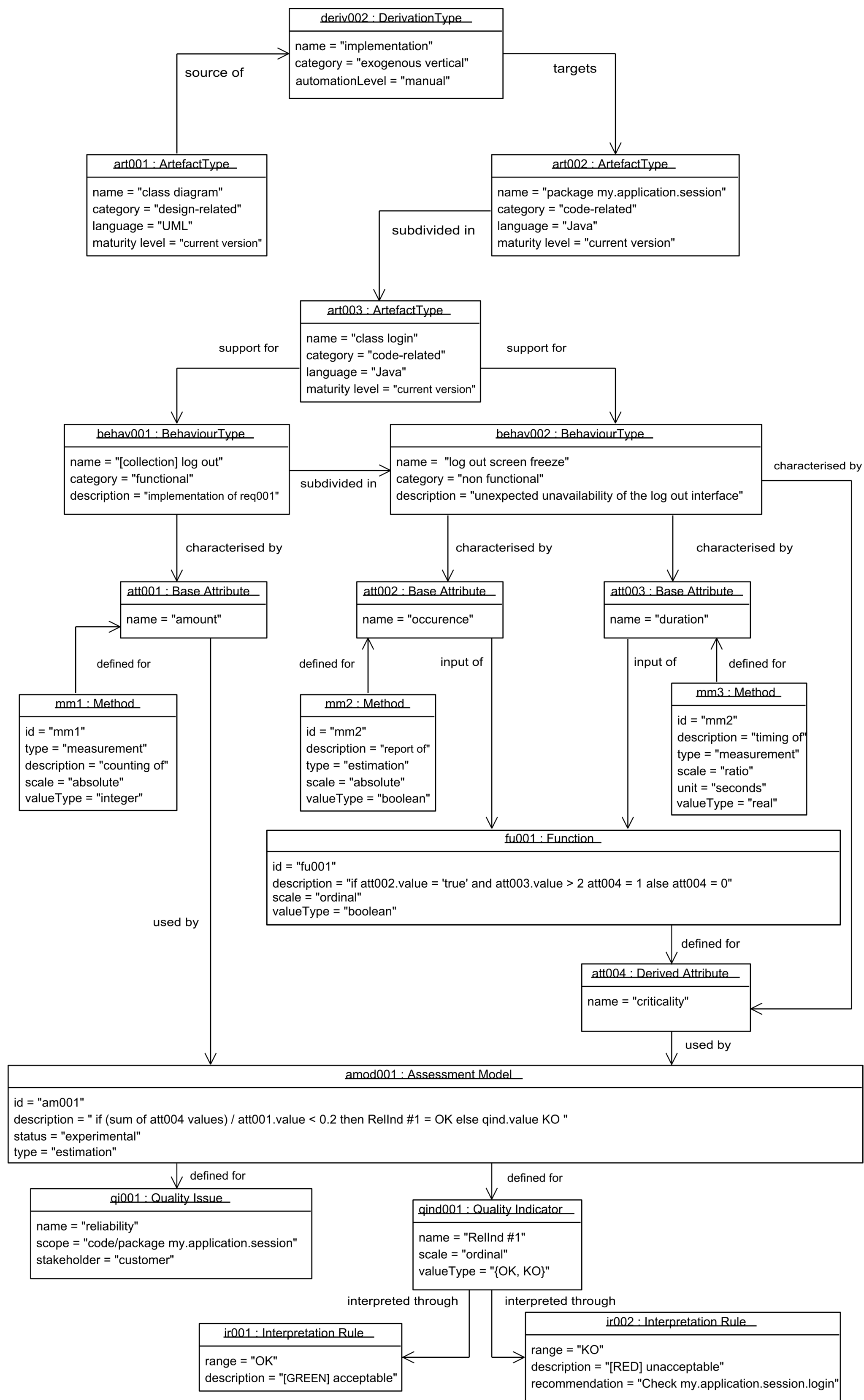

Figure 5.25: A complete (yet simple) example of MoCQA model

Attribute	Description
Range	Provides an interval of values (e.g., [0,20], [2.5,3.5[) or a specific value from a discrete set of values (e.g., “A”, “low”, etc.) for which the interpretation rule is applicable.
Description	Provides information on how to interpret the associated quality indicator for the associated quality issue if its value is comprised in the range.
Recommendation	Provides information on what actions should be carried out regarding the software project or the quality assessment process if the value of the associated quality indicator is comprised in the range.

Table 5.21: Attributes characterising an interpretation rule construct

Relationship	Description
interpreted through	Indicates the **quality indicator** for which the interpretation rule is relevant..

Table 5.22: Relationships involving interpretation rules

Illustration

Our example may be finalised thanks to interpretation rules as shown in Figure 5.25. Since the quality indicator `RelInd #1` may be assigned one of two values, we logically define 2 interpretation rules. The first is used when the quality indicator is assigned the ’OK’ value and associated this value with a green flag. No recommendation is provided since the reliability objective is supposed to be satisfied. The second interpretation rule is related to the ’KO’ value of the quality indicator and to a red flag, showing that the reliability objective is not completed. In that case, the recommendation of reviewing the code of the `login` class is provided.

This final addition completes a fully documented MoCQA model for a small example. It is worth mentioning that although this MoCQA model is structurally (or syntactically) well-formed, it is far from efficient regarding the quality assessment it models. For instance, the model displays a loss of definition and preciseness of the measure since it uses measurement methods providing “absolute” scales to finally provide a quality indicator that is associated with a “nominal” scale. This sort of flaw detection is further discussed in Chapter 8.

5.3 Designing the MoCQA model

As explained in Chapter 4, MoCQA models are fundamentally instances of the quality assessment metamodel. The instantiation process is constrained, on the one hand, by the quality requirements collected during the acquisition step. In other words, the instantiation process is executed in order to model these quality requirements and the way they are assessed.

On the other hand, the instantiation process in constrained by the methodological principles of the approach. Due to the fact that the MoCQA framework implements a top-down goal-driven methodology, the instantiation process is not a simple mapping between the quality assessment metamodel and the MoCQA model. The instantiation process has to performed as an ordered set of operations. Besides, provided that MoCQA models may be regarded as extended quality models, the rules defined to design a customised quality model (in [Dromey, 1996]) also apply to the design of a MoCQA model.

The Software Quality body of knowledge mostly agrees on the fact that quality goals should be defined first. This is obviously the case for the Goal/Question/-Metric approach. This is also the case in the five steps of approach proposed in [Dromey, 1996]:

1. Identify a set of high-level quality attributes for the product like reliability or maintainability.

2. Identify the product components. Examples are modules, requirements or relations.

3. Identify and classify the most significant, tangible, quality-carrying properties for each component. These are properties that result in manifestation of the high-level quality attributes.

4. Propose a set of axioms for linking product properties to quality attributes. This is not an easy task and the links cannot always be empirically verified.

5. Evaluate the model, identify its weaknesses and refine it.

As explained before, MoCQA models are built upon 3 components (i.e., assessment-related, measurement-related, project-related). According to the previous considerations, the instantiation process should define the quality goals first (and therefore, instantiate the assessment package to provide an assessment component), then the measurable entities (and therefore, instantiate the project package to provide a project-related component) and, finally, the measurement and/or estimation methods (and therefore, instantiate the measurement package to provide a measurement component).

5.3.1 Components instantiation

Concretely, the instantiation step consists in:

1. defining a quality issue (and possibly) its underlying hierarchy
2. defining the type of artefacts, derivations and behaviours of interest
3. defining or selecting the measurement methods to be used in this assessment or empirical study
4. iterating with a new quality issue until all available quality requirements are modelled

At each step of this process, the work-in-progress MoCQA model must be checked with regard to the abstract syntax defined by the quality assessment metamodel. The nature of the verification process is detailed in Section 5.3.2.

Additional methodologies may be used to strengthen this process. For instance, the quality assessment metamodel has been derived from the software quality ontology. The ontology itself has been designed so that no incoherence exists between its structure and the conceptual model of GQM/MEDEA. As such, the quality assurance team may rely on the GQM/MEDA methodology in order to ensure that the definition of the measurement process is defined properly while formalising the process into a MoCQA model. As explained before, the fact that quality issues are semantically compatible with the corporate objectives and the tactical and measurement goals provides a way to bridge the gap between the two methodologies.

[Kaner et al., 2004] provides a good starting point for an analysis grid that would help the user select the right (suitable) measures for a given purpose while not avoiding important characterisation about the measure (e.g., scale, unit, etc.).

Finally, note that the process of instantiation may be slightly altered in specific circumstances. For instance, defining the project component first, followed by the measurement and assessment components, may be regarded as a more "opportunistic" and ad hoc planning. It means that the user checks what measurable entities she may provide, tries to identify measures that could be applied and to infer what quality assessment she can derive from the measures. Although this method stirs away from pure goal-driven measurement, it is still not as unproductive as a real bottom-up measurement plan. Indeed, since MoCQA models are conceptual models that help plan the quality assessment, the goals are still defined before any actual measurement is performed, avoiding a waste of time and effort.

5.3.2 Structural coherence

During this process, some precautions must be taken in order to guarantee the overall robustness and integrity of the designed MoCQA model. The remain-

Construct	Attributes
Quality issue	Name Scope Stakeholder
Assessment model issue	Id Type Description
Quality indicator	Scale ValueType ValueRange
Interpretation Rule	Range Description
Base/derived attribute	Name
Method	Id Type Description Scale ValueType ValueRange
Function	Id Description Scale ValueType ValueRange
Artefact Type	Name Category
Behaviour Type	Name Description
Derivation Type	Name Category Multiplicity

Table 5.23: Mandatory attributes

der of this section addresses the structural coherence verification that has to be performed to guarantee the syntactic validity of MoCQA models.

Attributes

Although they may be completed with a lot of additional information, MoCQA models should at least provide a minimal amount of information in order to be exploitable. Table 5.23 provides a list of attributes that should be instantiated in any case. The attributes presented in the table are required in order to actually provide a measurement plan that is consistent and to allow the validation of the

MoCQA model during the exploitation step (see Chapter 8).

As we may see, quality issues require a name that encompasses the quality goal or requirement. Since it may apply to various elements, the scope is required. In order to guarantee that the quality issue is relevant, it should also provide an associated stakeholder.

Assessment models, functions and methods share the requirement to provide a description to make them applicable (since the description provide the core information of these concepts). Additionally, they must be assigned an id that must be unique since it will be referred to in the next steps of the methodology. Assessment models and methods also have to be characterised by a type in order to provide information on the overall rigorousness of the quality assessment process.

Methods and functions also characterise measurement or estimation values. As such, they are required to provide a scale, value type and value range, like quality indicators. Note that the value range may be ignored if the value type is an enumeration of values.

All measurable entity types have to provide a name which is their fundamental characteristic. However, the additional information required varies slightly from one to another. For derivation and artefact types, the category is more relevant (it provides a way to ensure that the derivation is adequate regarding the associated artefacts and a way to validate that the measured entities are coherent with the scope, respectively). The name of behaviour types may not be expressive enough by itself and therefore requires a description. The derivation type should also define an adequate multiplicity in order to allow the correct identification of the number of artefacts involved in the transformation.

Finally, interpretation rules have to provide a range and a description in order to be usable. As seen in previous examples, the recommendation may be ignored.

Associations

Regarding the associations of the MoCQA model, the quality assessment metamodel provides the main constraints that need to be validated. For instance, the quality assessment metamodel states that an attribute may be associated to at most one measurable entity type whilst a measurable entity type may have one or more associated attributes.

However, some precautions must be taken in order to guarantee that the MoCQA model remains acyclic. The "subdivided in" relationship allows this kind of cycle. Cycles may therefore arise between, quality issues, behaviour types and artefact types. Cycles are not relevant for any of those concepts and therefore, the process should ensure that the target of a "subdivided in" relationship is neither the source construct itself, nor one of its ancestors.

Derivation types and assessment models may also be used to create cycles in the MoCQA model. Similarly, a cycle is not relevant in that context. The design process should ensure that the target of any derivation type is not a (direct or indirect) source construct for the derivation type. Assessment models have to comply to the same constraints.

Chapter 6

Step 1: Acquisition

6.1 Overview

As explained in Chapter 4, the acquisition step focuses on the elicitation of relevant contextual information. This information concerns the software development environment in which the assessment process will occur. The information collected during this step also constitutes the input required to perform the quality assessment modelling of step 2 of the assessment methodology. In other words, the acquisition step ensures that the MoCQA model will actually be a *customised* quality assessment model.

The information collected can be divided in two categories. First, it contains elements of the environment that will not be explicitly modelled as part of the MoCQA model (e.g., budget constraints, time constraints, etc.). This information constitutes a reference for the quality assessment modelling process.

More importantly, the information collected at this stage is also comprised of elements that will be included in the MoCQA model explicitly, that is, the actual quality requirements.

6.1.1 Activities

Concretely, this step of the assessment methodology addresses the tasks described in the remainder of this section.

Planning of the quality assessment life-cycle

This task consists in defining how the quality assessment cycle will be performed with regard to the course of the software life-cycle. Since the MoCQA methodology may be used in a variety of contexts (e.g., in a development starting from scratch as a continuous supporting process, as a punctual process dedicated to the

profiling of an ending project, as a support for the maintenance process, etc.), it is important to define clearly how the quality assessment life-cycle will integrate into the development process. In other words, this activity is performed to define the objective of the introduction of the MoCQA framework.

Additionally, the type of software development life-cycle, together with organisational goals, mainly defines when quantitative results are needed, and the frequency of the successive assessment steps.

Identification of the stakeholders

This task consists in identifying relevant stakeholders to include in the acquisition process *for the current* quality assessment cycle. The rationale behind this task results from the fact that the MoCQA methodology is designed to avoid useless effort. As explained in [Westfall and Road, 2005], *" if a metric does not have a customer, it should not be produced".* Since measurement and assessment are time and effort consuming processes and before any value is collected or interpreted, the quality assurance team should make sure that the measure or indicator satisfies a need of at least one stakeholder.

However, due the iterative nature of the methodology, it would be counterproductive to try to include all possible stakeholders at the same time. This would result in long interviews and conflictual information. In order to avoid any overhead, only the stakeholders that will be involved in the current quality assessment cycle are selected. If new needs arise or new relevant stakeholders are identified due to the results of the current quality assessment cycle, it is always possible to come back to this step and refine the list of stakeholders consulted for the process. Note that methods inherited from Requirement Engineering (such as the guidelines described in [Sharp et al., 1999] may be applied to help identify the correct stakeholders at a given stage of the process.

Beside the identification of stakeholders, classifying them according to their types of needs may facilitate the quality requirement elicitation. [Westfall and Road, 2005] identifies 6 categories of "metric customers" (i.e., stakeholders in the MoCQA framework) that may be used as a guideline to perform the classification. These categories are the following:

Functional Management: Interested in applying greater control to the software development process, reducing risk and maximizing return on investment.

Project Management: Interested in being able to accurately predict and control project size, effort, resources, budgets and schedules. Interested in controlling the projects they are in charge of and communicating facts to their management.

Software Engineers/Programmers: The people that actually do the software development. Interested in making informed decisions about their work and work

products. These people are responsible for collecting a significant amount of the data required for the measurement program.

Test Managers/Testers: The people responsible for performing the verification and validation activities. Interested in finding as many new defects as possible in the time allocated to testing and in obtaining confidence that the software works as specified. These people are also responsible for collecting a significant amount of the required data.

Specialists: Individuals performing specialised functions (e.g., Marketing, Software Quality Assurance, Process Engineering, Software Configuration Management, Audits and Assessments, Customer Technical Assistance). Interested in quantitative information upon which they can base their decisions, finding and recommendations.

Customers/Users: Interested in on-time delivery of high quality software products and in reducing the overall cost of ownership. Additionally, they may be involved in the data collection by reporting defects or unexpected behaviour (i.e., through crash, bug or failure reports).

However, any type of classification may be applied if it fits the specific context (see Chapter 15 for a practical example of customised classification of stakeholders). For instance, the selected stakeholders target may be a specific team in charge of a component that is central to the software project, or individuals with specific concerns.

Environmental Constraints

This task consists in determining non-quality-specific properties of the environment. These properties will have an impact on the way MoCQA models are designed, on the way the assessment is performed and on the way the quality profile is interpreted.

Among the information that has to be collected through this activity, the clear identification of what types of resources are available may be the most important. Knowing what type of resources are available and the level of formalisation that is used throughout the development life-cycle (e.g., *"are the requirements formalised?"*, *"is their an important number of design-related resources produced?"*, etc.) will determine the type of measurable entities available and thus ensure that the quality issues are effectively assessable.

Other information may be useful to collect at this stage. Budget-related constraints and time-related constraints may help prioritize the quality issues. In the case of a maintenance process, information on the amount of time the maintained application has been used may be helpful to provide assessment models that take the age of the application into consideration.

In consequence, this step consists in the familiarisation with the environment in order to streamline the rest of the process. As for the identification of stakeholders, missing information is not irreversible and may be corrected in the next quality assessment cycle, if needed.

Quality requirement elicitation

This task represents the core of the acquisition step. It consists in the collection of quality requirements for each identified individual of group of stakeholders. These quality requirements, associated to specific stakeholders, will constitute the basis for the modelling of quality issues and subsequent elements of the MoCQA model.

Quality issues are also prioritized through this activity. Priority levels may be compared among stakeholders in order to start shaping the common understanding of quality issues for the project, resulting in a first alignment of the stakeholders' specific issues.

6.1.2 Formalisation

Although it is crucial to the efficiency of the quality assessment process, the acquisition step remains a rather informal step of the assessment methodology. The decision to leave this step informal is mainly due to the fact that the framework aims to remain flexible and adaptive. In order to adapt to the context in which it is exploited, the assessment methodology has to cope with the existing "organisational culture". The acquisition step involves several stakeholders of the organisation and as such, has to adapt to the set of procedures that apply in this environment. The output of this step could thus take the form of a report complying to a given template or remain semi-formal, depending on the environment.

6.2 MoCQA model design in practice

Chapter 5 investigated MoCQA models design from the theoretical point of view, that is, through their formal definition, abstract syntax, notation and so on. The theoretical definition of MoCQA models is sufficient to allow the exploitation of the framework as a support for the investigation of diverse theoretical research questions (e.g., the design of empirical studies such as in [Mens et al., 2011], the alignment of different norms and standards, the migration from one quality model to another, etc.).

However, the design of MoCQA models raises practical issues that have to be considered as early as the acquisition step. Indeed, MoCQA models are merely a formalism to structure the elements that constitute the quality assessment process, and make them available. The successful quality assessment of a software

project depends on the content of the MoCQA model, and therefore, on the relevance of the quality requirements elicited during the acquisition step.

This careful attention paid to the content of MoCQA models is especially crucial in a professional environment. In this case, the acquisition step represent a way to bridge the gap between the knowledge and expectations of the stakeholders and the assessment methods and techniques available to the quality assurance team. If the gap that separates these two points of view (i.e., what stakeholders expect and what quality assurance can offer) is bridged correctly, the quality assessment will be efficient.

The aim of the acquisition step is therefore to adapt the vocabulary and focus the process of designing MoCQA models. The framework relies on theoretical principles that may not be common concepts for practitioners (e.g., metamodel with abstract concepts). Therefore, the quality metamodel would require important and unproductive training prior to any interaction with the members of the development team, even if these members do not use the framework by themselves.

To solve these problems, the acquisition step represent an additional layer to improve the the usability of the framework in specific contexts (and in particular in an industrial context). That is why providing more effective ways to interact with the development team is beneficial at this stage. Instead of forcing the theoretical concepts the framework relies on into the context, the methodology acquires the information needed to design MoCQA models in a more operational way before using the metamodel as a filter for this information.

The following sections explore several acquisition methods that may be considered to provide this additional layer of support for the acquisition step.

6.2.1 Customising existing quality models

Translating a given quality model into a MoCQA model is the basic function of the quality assessment metamodel. Therefore, the direct customisation of an existing quality model is the most straightforward way to bridge the gap between stakeholders and quality assurance teams, provided that the quality model is well-known (i.e., the ISO/IEC quality model).

As explained before, [Wagner et al., 2009] shows that operationalisation of quality models is a complex and non trivial process. In the context of the MoCQA framework, however, this process may be regarded as an opportunity. As explained in Chapter 2, the problem with quality models is that they define a static view of what quality is supposed to be for a given product. On the contrary, the MoCQA framework is goal-driven and requires references in order to avoid defining goals from scratch. Quality models provide a useful source of quality factor hierarchies that can be translated into hierarchies of quality issues thanks to the quality assessment metamodel.

As such, quality models may be regarded as catalogues of quality factors that may be the centre of interest of quality goals to include in a MoCQA model as quality issues. The advantage of using a MoCQA model to support this customisation process is the fact that a systematic operationalisation template is applied to the quality models. For each quality factor, the quality assurance team has to investigate if the factor is relevant for at least one stakeholder in order to translate the factor into a quality issue. In consequence, the customisation is not performed in an arbitrary way but as a stakeholder-driven process. The process may be beneficial for the stakeholders themselves. Since quality factors are generally organised in a hierarchy, the use of a quality model may draw the attention of stakeholders on hidden sub-factors they may not have taken into account. For instance, if a given stakeholder expresses the need for portability, she may not be aware of the dependency to the co-existence factor defined in ISO/IEC quality model.

The other advantage of this acquisition method is the possibility to reuse the measurement methods defined in the quality model (if any are defined) and therefore facilitate the following steps of the MoCQA model design. If the measures defined in the quality model are not considered relevant, the quality assurance team may at least acquire a "template" for their following search for an adequate measure (i.e., the attribute they should measure, the scale in which the measurement values should be comprised, etc.).

Chapter 11 addresses the potential of several quality models to support a quality-model-based acquisition step.

6.2.2 Developing analysis grids

In order to assist the acquisition of the information needed to design a MoCQA model (on which the quality metamodel will be used as a filter), a viable solution is the design of a series of analysis grids taking the form of questionnaires. These grids may be used to analyse available project documentation and to interview the identified stakeholders. The main challenge is to provide questions that are cleverly designed in order to ease the design of the MoCQA models. The risk is to define questions resulting in unstructured information that would be incompatible with the constructs of the metamodel and therefore unusable. In order to achieve this goal, three design rules for the questions have to be followed:

- Limit the number of open questions as much as possible;
- Avoid using specific concepts of the metamodel in the questions as much as possible;
- Adapt the question to the role (and expectations) of the interviewed stakeholders

For instance, while instantiating the quality package of the quality metamodel, we have to cope with the concept of quality issue. In the questionnaire design, questions such as *"what are the quality issues for each stakeholder in your project?"* should be avoided. Instead, it would be more efficient to go to a specific stakeholder and propose different alternatives (e.g., *"is user satisfaction more important to you than maintainability?"*, *"would you rather focus on the portability of your project than efficiency?"*, etc.). This kind of questionnaires helps the quality assurance team associate the needs of the stakeholders to their own knowledge while providing a structure and a priority order for the quality issues.

Divergence in terminology may still arise with this process. The stakeholder may be used to refer to a concept of software measurement with a different denomination. Different techniques to refine the acquired information may be applied to align the terminology used in the context with the terminology of the literature. Typically, techniques designed to resolve semantic redundancies and ambiguities (typically in the human-computer interfaces reverse engineering field [Ramdoyal et al., 2010]) may be applied to our context. This way, if unexpected elements of terminology arise during the interview, it can be resolved and aligned with existing concepts of Software Quality. For instance, if the stakeholder expresses the need for its system to be *sustainable*, semantic similarities may indicate that the quality issue involved is in fact the *maintainability*. In the same way, a stakeholder asking for a minimum amount of bugs may in fact be talking about robustness.

Although the analysis grids may be an interesting option, their drawback lies in the difficulty to establish a suitable questionnaire. Effort should be made to produce a generic analysis grid that may be adapted to a specific context, in order to avoid the overhead of designing a grid for each context.

6.2.3 Tool support

As we have seen, one of the main challenges of the acquisition and design steps is to find the relevant knowledge to populate (while integrating the practical knowledge and expectations stakeholders already possess) the MoCQA model. Chapter 1 showed that a lot of analytical and declarative methods exist. The choice of adequate elements to include in the MoCQA model may therefore be complex.

One response to this problem is to provide a knowledge base to support the acquisition of suitable measurement/estimation methods based on specifics queries (e.g., "what measurement method could produce a value that is comprised in a ratio scale so that it complies to the quality indicator we have to provide for this quality issue?").

Due to its ontological nature, the MoCQA metamodel can be used as a basis for the creation of such a knowledge base. Chapter 9 describes an attempt (the

QuaTALOG project) to provide such tool support. Although any sort of quality-related repository may be profitable to the acquisition and design steps, the use of a knowledge base relying on the MoCQA quality metamodel as an ontology should help focus the search for related quality concepts that are easy to integrate in a MoCQA model.

6.2.4 Complementarity with scenario-based analysis

So far, the focus of this dissertation has been mainly limited to quantitative approaches (or metric-based approaches) to software quality assessment. However, different approaches to software quality assessment have been proposed. Among these methods which adopt a different take on quality assessment, scenario-based software architecture analyses [Koziolek, 2011] are compatible with the MoCQA approach.

While, metric-based approaches rely on software measurement and focus on quantitative assessment of software architecture, scenario-based approaches to software architecture quality assessment adopt more participative methodologies. Their efforts focus on the elicitation of precise and manageable quality requirements thanks to scenarios designed in collaboration with the various stakeholders. Besides, these methods rely on an explicit description of the architecture [Bengtsson et al., 2002]. Many scenario-based evaluation methods have been proposed [Clements et al., 2001].

Although they are promising, the scenario-based methods still lack validation on their potential return on investment. More importantly, existing scenario-based methods and metrics are not yet integrated together although their complementary takes on quality assessment could result in a more efficient and thorough evaluation of software architecture [Koziolek, 2011].

The focus on eliciting quality-related requirements is typically adapted to the acquisition step of our approach. Efforts carried out to conduct a scenario-based analysis may be formalised as MoCQA models and therefore provide a bridge between elicitation and measurement. This acquisition method is explored in Chapter 12.

6.2.5 Complementarity with Requirements Engineering

The acquisition methods considered so far deal mainly with the integration of knowledge from the literature in a meaningful way for the stakeholders. This method is more concerned with the formalisation of existing and practical knowledge from the environment.

As explained before, the constructivist perspective of the framework may be regarded as the possibility to “implement” quality from a MoCQA model in the same way code is implemented from design. Therefore, quality requirements may

be acquired with comparable techniques to those used in Requirement Engineering.

As for a regular requirements elicitation process, quality assurance teams relying on the MoCQA methodology have to take both the expectation and the experience of the stakeholders into account. For instance, stakeholders may have acquired some experience regarding how to achieve and monitor specific quality goals in previous projects. Similarly, some members of the team may have developed good practices over time the other members may not be aware of. This experience may be formalised and objectified through the MoCQA model. Requirements elicitation frameworks such as the framework proposed by Software Engineering Institute [Christel and Kang, 1992] are applicable to this step.

Among the complementary efforts that may be used to strengthen the acquisition step, research addressing the elicitation of non-functional requirements may be particularly useful. [Miller, 2009] proposes over 2000 questions focusing on the elicitation of non-functional requirements (i.e., quality requirements). These questions may be used independently or in conjunction with the analysis grid method described in Section 6.2.2.

Goal modelling is also a complementary method when it comes to the acquisition of quality requirements to populate a MoCQA model. Goals are *"declarative statements of intent (to be achieved by the system under consideration)"* that *"may refer to functional or non-functional properties"* [van Lamsweerde and Letier, 2004]. Goal models may be used to structure these goals and, notably, identify conflicts [Yu and Mylopoulos, 1998]. Non functional goals from a goal model (i.e., safety, fault tolerance or security) may therefore easily be translated into quality issues and associated to evaluation methods designed to assess their satisfaction.

Elicitation tools may also be used to complement the tool support. Among them, ElicitO [Al Balushi et al., 2007] relies on a underlying ontology that is compatible with the MoCQA quality assessment metamodel. Although this ontology is less detailed than the quality assessment metamodel, it provides the basic structure (i.e., characteristic, sub-characteristic and measure) of a quality model. Any quality requirement collected through this tool may be translated into a hierarchy of quality issues.

Finally, the field of creativity-based requirement engineering offers interesting opportunities for the acquisition step. [Mich et al., 2004] introduces the EPMCreate process which intends to improve the effectiveness of traditional brainstorming. This process, based on a model of the pragmatics of communication, regards each step of the elicitation process as suggesting a way for an analyst to look at the problem from a different stakeholder's viewpoint. The process allows for a systematic exchange between stakeholders, where propositions from each stakeholder are reviewed and possibly improved by other stakeholders. As we have

seen, the constructivist approach adopted by the framework induces the need to construct a common agreement of what quality is for the project and how it can be achieved. This exchange of problem elicitation and solution proposals may be beneficial to the acquisition step. These kinds of techniques are especially valuable during the design of interpretation rules, since these concepts require more creativity than the definition of the quality issues. The proposal of actions associated to a given indicator value is highly sensitive to the correct interpretation of the results and would therefore benefit from an improved brainstorming method.

Chapter 7

Step 3: Measurement Plan

7.1 Overview

In Chapter 4, we defined the tailoring step of the assessment methodology as the bridge between the conceptual and operational levels of a given quality assessment cycle. As such, this step is concerned with providing the guidelines for the actual measurement and quality assessment based on the conceptual definitions provided in the MoCQA model. In other words, the tailoring step focuses on ensuring that the MoCQA model will actually be an *operational* quality assessment model.

Defining a measurement plan is a common step of traditional quality assessment frameworks. According to the definition from Chapter 3, the goal of the measurement plan is to organise the steps of the measurement process. In terms of MoCQA-related concepts, the measurement plan defines what **measurement or estimation values** to collect, thanks to what **measurement or estimation procedures** and from which **measurable entities**. It also defines how **assessment models** are formalised. Additionally, the definition of which person will perform which process should be addressed.

The introduction of MoCQA models in the quality assessment process provides a majority of this information but on a conceptual level (i.e., with a level of detail and formalisation that is mainly dedicated to the communication between individuals). This step is concerned with the concrete guidelines that allow the execution of a quality assessment that "implements" the MoCQA model.

7.1.1 Activities

Concretely, this step of the assessment methodology addresses the tasks described in the remainder of this section.

Adapting the MoCQA model

The main objective of MoCQA models is to be expressive and allow an easy communication between stakeholders. As such, the quality assessment metamodel provides constructs that help define easily complicated relationships between measurable entities. For instance, derivation types define a relationship between artefacts and allow the evaluation of this relationship.

The drawback of this approach is that some elements may not be actually measurable directly (e.g., if the derivation type is not formalised in a transformation language). However, MoCQA models, thanks to their explicitly defined metamodel, may undergo transformations that preserve the semantics of its elements. In consequence, the first task of the tailoring step may be to transform the MoCQA model so that it may be operationalised and applied through a concrete measurement plan. This task is discussed in Section 7.2.

Operationalising the MoCQA model

This task is the core of the tailoring step. It mainly consists in transforming the MoCQA model (which is designed to be conceptual and stakeholder-oriented) into a model that is closer to an actual measurement plan. In consequence, this step consists in providing the following information:

1. A practical way to identify resources that have to be measured
2. Measurement/estimation procedures associated to each measurement/estimation method
3. Actual assessment models (i.e., algorithms or formulas) based on the description provided in the MoCQA model (which is similar to the method-/procedure relationship)

The operationalisation of MoCQA models is investigated in Section 7.3.

Preparing the data collection

The last inherent task to the tailoring step is to provide an infrastructure to collect and keep track of the measurement and estimation data. The introduction of the MoCQA methodology induces some changes in the way data is stored. The repository designed to store the measurement data and quality indicator values has to co-evolve with the MoCQA model since the latter may introduce drastic changes as the quality assessment life-cycle unfolds (e.g., introduction of new measurement or estimation methods, deprecation of other methods, etc.). In order to avoid the systematic redesign of the repository, the framework proposes a generic conceptual schema for the design of repositories designed to support the application of the MoCQA framework. This schema provides a description of how data should be stored in order to take the evolution ot MoCQA models into

account while not redesigning the repository for each new version of the MoCQA model. Section 7.4 addresses this task and the conceptual schema it is supported by.

7.2 MoCQA model transformations

The level of expressiveness allowed by the quality assessment metamodel provides various syntactic options to express a given semantics. In other words, there are different ways of modelling the same notion within MoCQA models. As such, MoCQA models may undergo transformations that preserve the same overall meaning. As explained before, the tailoring step may benefit from such transformations in order to make the assessment step easier to perform.

During the course of this research, two types of transformations have been identified as relevant in the context of the operationalisation process: the removal of derivation types and the introduction of collections of measurable entity types. The remainder of this section investigates these two types of model transformations. Both types of transformation are strictly endogenous and horizontal (i.e., akin to a refactoring activity) according to the taxonomy of model transformations defined in [Mens and Gorp, 2006].

Introduction of collections of measurable entity types

As explained in Chapter 5, measurable entity types may be defined as "a collection of all entities". The rationale behind a "collection" entity type is to allow the accurate definition of attributes when no parent exist for a given measurable entity type. For instance, a "Java class" may be associated with a "size" attribute but the semantics of this association implies that for each Java class, the size of this specific class will be evaluated. If one wants to rely on the number of Java classes for one function or assessment model, the size attribute should be associated with a "package X" artefact type, since the number of classes contained in a package characterises the package and not the class. If no parent exists for the measurable entity (or in order to avoid the systematic use of an "artificial" parent such as "source code"), the same evaluation may be modelled as an assessment model computing the sum of all entities of this type through a size attribute associated to a "collection of all entities of type X". For instance, in the previous example, the formalism allow the definition of a "[Collection] Java class" artefact type which indicates that all Java classes are considered as a single measurable entity. This entity type behaves like any other measurable entity type. For instance, if the defined artefact type is the target of an "implementation" derivation type that is associated to a "UML class diagram X", the semantics of MoCQA models indicates that all Java classes implemented on the basis of this specific class diagram X are considered as a single measurable entity.

Another way to model the same process is to define a "presence" attribute (i.e., value of 1 associated to each existing entity of the type) associated to a "Java class" artefact type and compute the sum of the array of values in an assessment model. Although the results would be the same, many automated measurement procedures do not provide support for the evaluation of this occurrence attribute and generally consider the entire population of entities.

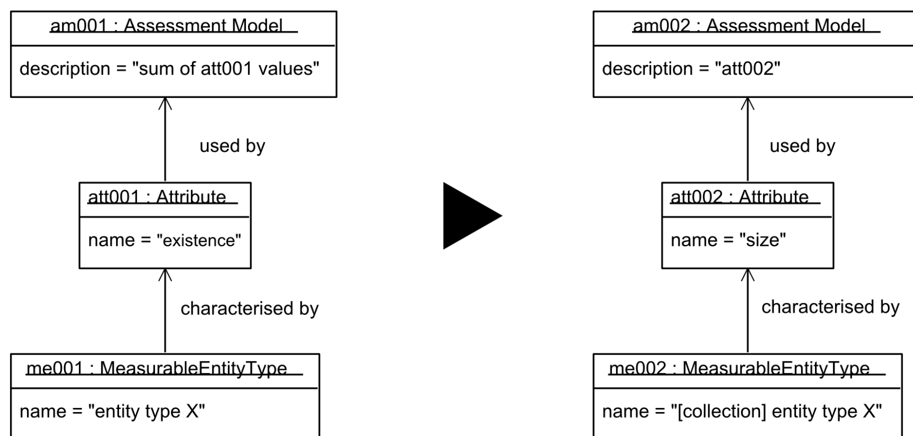

Figure 7.1: Introduction of a collection of entity type

In consequence, it may be beneficial to the operationalisation process to systematically perform the transformation shown in Figure 7.1. On the one hand, the semantics of the MoCQA model is preserved. On the other hand, the size attribute associated to the collection is more easily translated in a (semi-)automated procedure during the next task.

Removal of derivation types

Unless they are formalised in some way, derivations represent abstract relationships between artefact types. As a result, it may be difficult to evaluated associated attributes of derivation types. Conversely, artefacts are easier to locate, identify and measure.

In any case, derivation types may be regarded as additional constraints defined on the source and target artefact types, like the "subdivided in" association. For instance, an "implementation" derivation type defined between a "class diagram X" artefact type and a "Java class" artefact type states that the assessment will consider each Java class that has been implemented on the basis of said class diagram. As such, it is possible to remove a derivation type from the model and preserve the semantics of the model by adding constraints to the description of the involved artefacts.

As shown in Figure 7.2, we may remove a derivation type and embed the information provided by the derivation in the source/target artefact themselves. For

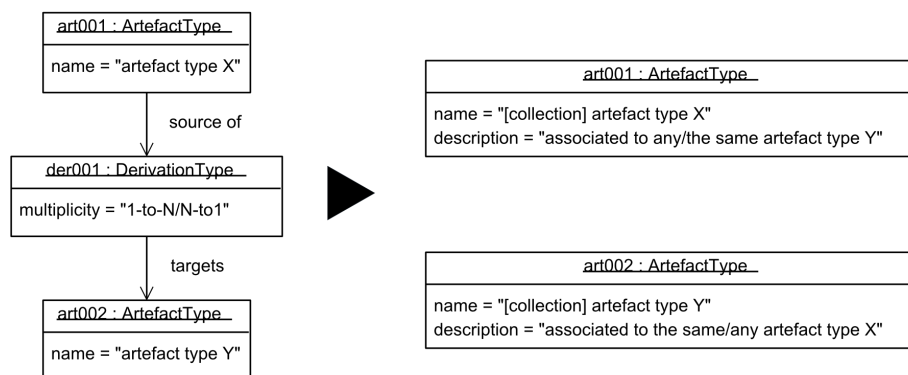

Figure 7.2: Removal of a derivation type

instance, let's consider the "documentation" derivations that occurred between "Java class" artefacts and "Documentation sections" artefacts. The multiplicity of this derivation type is set to "1-to-N" since a single class may require several sections of documentation. One may be interested in counting the number of occurrences of this derivation type (e.g., in order to compute the total number of classes that have been covered by the documentation process). Counting derivation types may not be easy unless a log of the action performed is kept throughout the development process. In that case, applying the transformation described above may help understand what elements have to be counted exactly. Applying the transformation, the derivation type is no longer present but the target artefact type is now defined as the collection of all documentation section associated to a same Java class. Based on this definition counting the number of *occurrences* of such collections provide the same number as counting the number of class that have been documented.

Although this type of transformations tends to make the MoCQA model itself more difficult to read, it provides a more accurate definition of the involved artefacts. It is therefore adapted in the context of the measurement plan but is not recommended during the design step. An illustration of such operationalisation is provided in Chapter 13.

7.3 Operationalisation challenges

As explained before, the operationalisation of MoCQA models consists in transforming the model so that it retains the same information, augmented by elements that help bring the model closer to an actual measurement plan. These elements are related to three types of details :

1. Details provided to help identify the resources that have to be measured

2. Details on the measurement/estimation procedures associated to each measurement/estimation methods
3. Details on the actual assessment models (algorithm/formula) based on the description provided in the MoCQA model

In many regards, the operationalisation of a MoCQA model may be seen as model transformation akin to the implementation. As in the case of an implementation activity, the information of the design (MoCQA model) has to be translated in a formalism that is more specific and has to provide a more detailed version of the information (a truly operational customised quality assessment model).

Regardless of the formalism used to provide the measurement plan, the operationalisation is therefore an exogenous and vertical model transformation since the metamodels of the two models are distinct (i.e., the quality assessment metamodel for the MoCQA model and metamodel that defines the formalism selected to provide the measurement plan) and the level of detail is increased. Each component of the MoCQA model calls for specific additional details.

Assessment component

During the operationalisation step, assessment models are the main focus regarding the assessment component of MoCQA models. The objective of the operationalisation regarding assessment models is to provide a definition of the model that is more structured and may be automatically computed. The definition builds on the more generic description provided during the previous step.

This definition is therefore a script written in a specified language selected by the quality assurance team so that it may be adapted to specific tools or organisational procedures. The only constraint regarding the content of the script is that the "id" of an input attribute must be used as variable names (i.e., typed as array of values, according to the semantics of MoCQA models). The "id" of the output derived attributes are also used as variable names.

Measurement component

As explained before, the core of the operationalisation task is to define a suitable measurement procedure to "implement" the method. This task therefore concerns the measurement component. During the development of the framework, we identified the following 4 relevant types of possible evaluation procedures in the context of the framework:

1. Measurement procedures performed manually (for instance, COSMIC-FFP)
2. Measurement procedures performed automatically through a specified tool (for instance, Coupling using SDMetric[1])

[1] http://www.sdmetrics.com/

3. Data mining in a repository
4. Manual inspection (done by a specified operator)

Manual procedures rely on the definition of measurement procedure (i.e. *"A set of operations, described specifically, used in the performance of particular measurements according to a given context"*) and are therefore an ordered series of instructions provided to the individual in charge of the measurement. Automated procedures only require the specification of a tool name and a metric provided by this tool. Procedure relying on the data mining of a repository have to specify the name of the repository and a query to perform (in a defined language). Finally, manual inspection procedures only need to provide the list of the reviewers assigned to this task. Finally, the operationalisation of the measurement component also calls for a formalisation of the functions used to provide derived attributes. Similarly to the definition of the assessment models, functions may be operationalised through the declaration of a scripts written in a specified language. The only constraint on the content of the script is that the "id" of input and output attributes must be used as variable names.

Project component

Artefact types, derivation types and behaviour types are basically translation of their MoCQA counterpart in the measurement plan. The details required at this level may be provided through a set of metadata defined to ease the search of the relevant artefacts (or derivations if they are formalised in a specified formalism) in the set of actual resources of the project.

For instance, metadata formalising the search for a specific artefact type (or derivation) based on a regular expression and additional keywords may be used in the measurement plan. This metadata bridges the gap between the conceptual level of the framework and a (possibly) automated measurement process.

Behaviours do not need to be located per se and therefore do not require an operationalisation step (although the description has to be clear enough to allow the observation of the described behaviours).

7.3.1 Formalisation of the operationalisation

The information pertaining to the operationalisation step (and the inherited information from the MoCQA model) has to be presented in a formalism that is easy to store, manipulate and revise due to the iterativity of the MoCQA methodology. In order to support the operationalisation process, the framework provides an XML-based language (XOCQAM) designed to express MoCQA models and capture the additional information that relates to their operationalisation, as well as facilitate their persistence. This formalism is further discussed in Chapter 9.

7.4 Preparing data collection

The collection of measurement and assessment data in the context of the MoCQA methodology has to cope with two specific challenges. First, the types of measures are predefined but specified by the MoCQA model itself. On the other hand, MoCQA models evolve. Therefore, the interpretation of a given measure collected at a given time has to refer to the MoCQA model it is based on.

In order to avoid the systematic redesign of the measurement and assessment data repository, we have to define a more generic conceptual schema for the data collection. The process is general and may be implemented according to the organisational requirements (e.g., through a database management system, a set of XML files or even a simple excel document), as long as the repository complies to the conceptual model. The remainder of this section addresses this conceptual schema.

7.4.1 Data Model

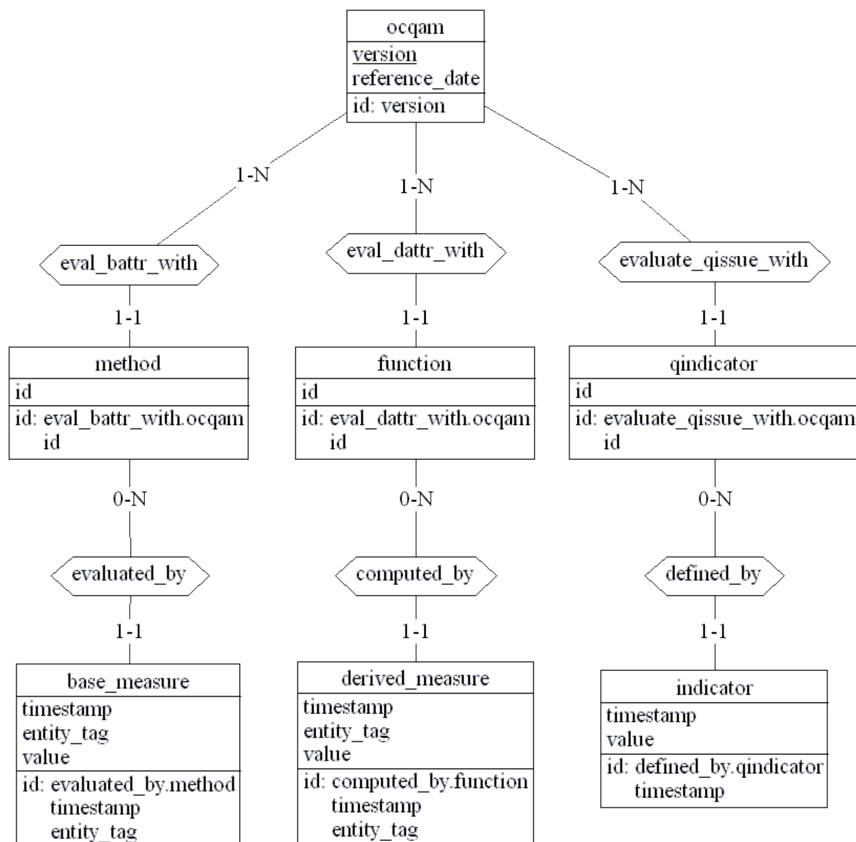

Figure 7.3: The MoCQA data model

The data model introduced in Figure 7.3 shows that the data collection has to provide a strong relationship with the MoCQA model/XOCQAM document that supports the current quality assessment cycle. Therefore, the data model is designed to keep track of two things. First, it obviously allows the persistence of the actual measurement and assessment values. However, it also contain information on selected elements of the MoCQA model, as well as on the model itself.

Regarding the measurement values, we may store their information as alphanumeric data, as long as the repository allows the traceability with the elements of the MoCQA model that defines the actual type of the values. Therefore, the conceptual schema includes information on methods, functions and quality indicators since these constructs characterise the measurement and assessment values in the MoCQA model.

In order to avoid duplicating the information that exists in the MoCQA model, the only piece of information required in the repository is the "id" attribute of these elements. However, this id attribute is not sufficient to clearly identify the content of these elements since they are bound to evolve through time. In consequence, the repository also has to collect information on the MoCQA model itself. This information simply consists in a version number that allows the identification of the correct content the quality assurance team has to refer to during the interpretation. Methods, functions and quality indicators are thus identified by their own "id", coupled with the version number of the MoCQA model.

Additionally, a given method or function of a given version of the MoCQA model may be applied several times. The measurement/estimation values collected are also as numerous as the number of entities in the given entity population defined by the measurable entity type. Therefore, what makes a value identifiable in the context of the MoCQA methodology is the time and date at which the value has been collected, coupled with the precise entity it has been collected on (identified with a unique entity_tag) and the reference method or function the values has been produced with.

This conceptual schema of data persistence guarantees that the values are always linked to a specific version of the MoCQA model, as well as with the actual resources of the software project. This allows an efficient use of the repository, in conjunction with the MoCQA model that provides a meaning to these values.

Chapter 8

Step 5: Exploitation

Model-checking, Quality Profile and Decision-making

8.1 Overview

As explained in Chapter 4, the exploitation step is mainly a decision-making step. It concludes a quality assessment cycle and allows the preparation of the next cycle. The previous steps of the methodology contribute to the elicitation of quality requirements for the identified stakeholders, the elaboration of a plan to assess their satisfaction and the actual collection of measurement and assessment-related data. The exploitation step brings all these elements together in order to provide information to the stakeholders. This information is the basis of a decision-making process focused on two distinct aspects: the actions required regarding the software development process and the actions required regarding the continuation of the quality assessment process.

8.1.1 Activities

Concretely, this step of the assessment methodology addresses the 2 tasks described in the remainder of this section.

Quality profiling

Quality profiling consists in mapping the actual data collected during the assessment step with the information contained in the MoCQA model. The latter provides information designed to interpret adequately the measurement values collected on multiple elements of the software project.

Quality assessment refinement

This task consists in reviewing the MoCQA model designed in the second step of the methodology. This reviewing process may be performed before the measurement plan is actually tailored and applied (since step 3 and 4 may be ignored). In the other case, the reviewing process may be performed following inadequate or controversial results from the profiling activity. In any case, this activity allows the quality assurance team to discover and correct mistakes in the quality assessment process, or to add new quality issues if the quality profiling of the project reveals new information needs expressed by the stakeholders.

8.2 Quality Profiling

The fourth step (assessment) of the methodology described in Chapter 4 focuses on the actual measurement-related and quality data (i.e. measurement results) collection in order to produce a quality profile of the software project. This step (exploitation) relies on the quality profile that has been produced. Within the framework, the quality profile adopts a specific meaning and may be defined as follows:

Definition 8.1 (Quality profile).
The collection of all identified measurable entities contained in the project, their associated measurement or estimation values, as well as the values of the various quality indicators, their concrete interpretations and the underlying definitions of the MoCQA model.

The quality profile is exploitable in several ways. First and foremost, the quality profile includes actual values of the quality indicators and their meaning. The values are direct answers to the quality issues (i.e., the information needs of specific stakeholders). The most basic function of the quality profile is therefore to assess the current level of satisfaction of the quality objectives.

Secondly, since the quality profile includes the entire set of actual resources of the project associated with a measurement or estimation value, the quality profile may be used to analyse the causes of unsatisfactory assessment results and pinpoint adequately the elements that contributed to the poor results.

Finally, thanks to the underlying MoCQA model, even an incomplete quality profile (i.e., with no actual measurement data but with already identified measurable entities) may be used to guide the design and development of the software project and anticipate the way it will be assessed.

8.2.1 Interpreting quality indicators

For each quality indicator defined in the MoCQA model, a value complying to the indicator definition (i.e., scale, value type, value range) should be available, following the assessment step. Interpreting the quality indicator obviously requires the consultation of the adequate interpretation rule (i.e., the rule for which the actual value of the indicator falls into the defined range). This rule provides a way to attach a meaning to the quality indicator value. However, interpreting the indicators also requires the consultation of the attached quality issue. The quality issue provides information on what question the quality indicator seeks to answer, for which elements of the software project it is relevant and towards which stakeholders the answer should be directed.

The MoCQA model that supports the quality profile therefore helps the quality assurance team provide the right information to the right stakeholder. In their XOCQAM form, they also allow basic queries to increase the efficiency of the process (e.g., sorting out elements by scope or measurable entity types) and to offer different viewpoints on the quality profile.

Once the value of the quality indicator is associated with the suitable scope, stakeholder and interpretation rule, the quality assurance team may provide a targeted answer to the information need (i.e., the description of the interpretation rule) and investigate whether the action recommended by the rule is relevant. If the description or recommendation from the interpretation rule does not seem to comply to the stakeholder's expectations, new information needs may be added and may lead to a new assessment cycle. If the recommended action is selected and involves other stakeholders (typically members of the development team), these stakeholders should be consulted in order to assess the relevance of the recommended action. This course of action may lead to a root-cause analysis supported by the quality profile.

8.2.2 Supporting root-cause analysis

As explained in Chapter 4, the notion of explicit and integrated quality assessment modelling is expected to support the communication between stakeholders. Past the first interpretation of quality indicators, the information comprised in the quality profile may be used to support a diagnostic process similar to root-cause analysis.

Root-cause analysis is *"a process designed for use in investigating and categorizing the root causes of events with safety, health, environmental, quality, reliability and production impacts"* [Rooney et al., 2004]. Root-cause analysis is applied in various fields outside Software Engineering. In Software Engineering, the events targeted by the technique are mainly defects. The specificity of root-cause analysis is that it is designed to help identify why some defects arise

in addition to their mere detection. The rationale behind the method is that it is crucial to understand why a specific defect is detected in order to provide corrective actions that prevent future defects.

The quality profile obtained through the MoCQA methodology provides the fundamental information to support a lightweight or more thorough root-cause defect analysis. The key element of the quality profile that provides this support is the identification/tagging procedure performed during the assessment step. Indeed, MoCQA models explicitly bind the actual resources (e.g., the source code, a given diagram, a test case, etc.) to the associated quality issues. As shown in Chapter 7, the methodology integrates the preservation of this information (i.e., each measure taken). Therefore, it becomes easier to spot the artefacts that require improvement through the analysis of the values themselves and the support of the description of the assessment models they are used by. For instance, let's consider a quality issue that targets the complexity of the code and the associated assessment model that is defined as the **average of the complexity (attribute) of every Java class (i.e., artefact type = Java class** evaluated thanks to McCabe's cyclomatic number (method). If the quality indicator provides a value of 60 (which is pretty bad according to the definition of McCabe's cyclomatic number), we can track any identified/tagged Java class associated with a complexity of more than 60 and conclude that some of these specific classes may require a complete refactoring.

8.2.3 Exploiting MoCQA models during software evolution

MoCQA models may be exploited more in the context of the maintenance and evolution phase of the software development. The remainder of this section illustrates how MoCQA models may be used to support a selected number of software evolution challenges (from [Mens et al., 2005b]) and describes example applications of the framework that contribute to tackle these specific challenges. Chapter 13 describes a case study investigating how the framework actually performs when used to support the maintenance and evolution of software.

Preserving and improving software quality

According to [Parnas, 1994] and [Lehman et al., 1997], software systems that are not carefully inspected from a quality point of view, see their quality gradually decrease as the systems evolve. This need to constantly re-evaluate the quality aspects of an evolving system is a key aspect of the MoCQA framework. The MoCQA model is a map of which quality issues are monitored, as well as the extent to which they are satisfied. Therefore, the MoCQA model is a direct roadmap for evolution. We can see what quality issues still require improvement or even if a desired quality characteristic has been tracked or not. As explained

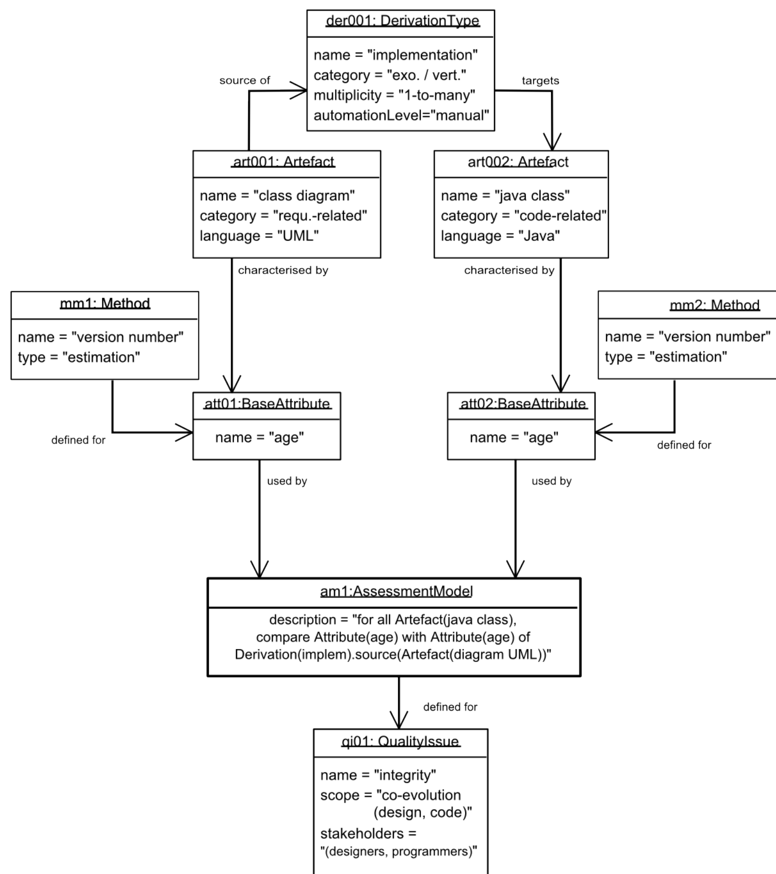

Figure 8.1: A simplified MoCQA model applied to co-evolution

above, the fact that each measurement/estimation value is preserved and associated with a specific resource (e.g., file, class , diagram, etc.) is also a valuable mechanism to obtain an accurate picture of what actions are to be carried out to improve the overall quality.

Supporting co-evolution

In any medium or large software project, many artefacts with different levels of abstraction are involved in the process (e.g., UML diagrams, code, etc.). The challenge of co-evolution is to reflect the modification of a given artefact to any related artefact in order to guarantee the consistence of the project. Thanks to the explicit and integrated quality assessment modelling, the integrity of co-evolution may be modelled as a quality issue, ensuring that the development team will focus on this aspect. As shown in Figure 8.1, the modelling of such a quality issue relies on the fact that it is associated to two types of stakeholders instead of one (i.e., programmers and UML designers). A basic assessment model for the quality issue would be to constantly compare the version (defined as an

estimation method for the "age" attribute) of related artefacts. However, more accurate assessment model could be defined on the basis of more specific metrics (e.g., number of classes, number of attributes in the class, etc.).

Support for multi-language

The challenge of managing a real software development includes the requirement to support more than one language at a time. The framework provides support for multi-language environment. In fact, this support is ensured by the MoCQA model that lies on a high level of abstraction. At this level of abstraction, language is just a characteristic of a measurable entity. It provides the maintenance team with a better understanding on how multi-language the project is and which stakeholder is associated with what language. As a consequence, this allows distinguishing explicitly what is comparable (e.g., measures defined on entities of same language, or one family of languages explicitly identified like object-oriented languages) from what is not comparable (measures defined on entities having no common super class).

Integrating change in the software life-cycle

A typical way to integrate changes in the software life-cycle is to rely on incremental and iterative development processes. Each iteration or increment is a shorter-lived cycle that helps maintain a good level of flexibility towards changes. The MoCQA framework is designed to help maintain a consistent quality assessment throughout successive iterations. First, MoCQA models ensure the traceability of quality, especially in an iterative life-cycle, allowing the monitoring of the evolution of quality (i.e., the continuous monitoring of quality) as well as the quality of evolution (i.e., how well evolution-related tasks are performed). Besides, the assessment methodology described in Chapter 4 supports the evolution of the quality assessment strategy itself.

Increasing managerial awareness

Thanks to the hierarchy of quality issues associated with specific stakeholders, a MoCQA model is a suitable mechanism to increase the managerial awareness of evolution needs. A MoCQA model explicitly links the quality issues with actual artefacts, derivations and behaviours, providing the managers with many explanations and precise information about the origins of the flaws. This information may allow a better planning (i.e., prioritisation) of the maintenance and evolution activities, directly linking the information with the impact on the quality issues that managers (as stakeholders) are interested in.

8.2.4 Exploiting MoCQA models at early stages of the development

MoCQA models may be exploited before the quality profile is actually completed. In that case, they fully achieve their goals as a guiding method. MoCQA models are available for any stakeholders including the development team and precisely describe the quality requirements for the project, as well as the way the quality team intends to assess their satisfaction. As such, once the measurable entities are identified, the development team is able to evaluate the impact of their actions on the overall quality of the software project. For instance, let's consider a quality issue that targets the maintainability of design and is assessed through the size (attribute) of the class diagrams (artefact type). The planned method is to evaluate the number of classes in the diagram and confront it to a given threshold (a method that is close to the predictive model found in [Bocco et al., 2005]). Having this information at their disposal, designers may try to reduce the number of classes when design choices allow for this reduction. If the threshold defined for the maximum size of the diagrams is unsustainable regarding the architecture they have to model, the designers may push the information back to the quality assurance team that may in turn review the expectations of the stakeholders and make them more realistic. Similarly, the development team may detect conflicting quality issues (i.e., requiring two opposite actions on the measurable entities) associated to the resources they are working on. In that case, they may detect and report possible failure of the quality assessment process, before it is actually performed. The priority level of the conflicting quality issues may then be used to determine which one should be considered first. An example of such an application of MoCQA models is described in Chapter 12.

8.3 Reviewing MoCQA models

MoCQA models may be reviewed during the exploitation step. This reviewing process may occur even prior to any evaluation. The tailoring and assessment steps may therefore be ignored if the quality assessment modelling is not yet completed and the aim of the current quality assessment cycle is to assess the MoCQA model itself (as explained in Section 4.3.2). This reviewing process consists in checking the integrity of their content and checking some properties of the models.

8.3.1 Content integrity

Chapter 5 provides the rules required to ensure to structural coherence of MoCQA models. However, those rules do not suffice to ensure that the content of the MoCQA model is consistent with the quality assessment it aims to model. Some additional coherence verification (addressing more semantic concerns) must be

performed in order to ensure that the quality assessment process modelled is sound and robust. Concretely, many entities of a MoCQA model are associated with other entities of the model. These associated entities have to be defined in a coherent way (i.e., the information encapsulated by this set of entities must not be conflictual). The verifications performed on the MoCQA model consist in checking that these associated elements are correctly defined themselves and in conjunction with the other elements that are part of the association. The remainder of this section describes the verifications that have to be performed before the MoCQA model is applied. Note that these verifications do not guarantee that the assessment process is correct (i.e., that the measurement performed actually relies on accurate and suitable measurement and assumptions) but that the MoCQA model is exploitable once the assessment step is performed. The correctness of the content depends entirely of the acquisition step (where quality requirements are elicited) and the design step (where measurement are selected or designed, possibly through more rigorous methodologies).

Scope and categories consistency

The category of measurable entities used to provide input values to a assessment model should always be related to the scope of the quality issue (e.g., code-related entities for the "source code" scope). This relation does not need to be direct. For instance, one may attempt to assess the maintainability of the code, based on the size of the design. In that case, the measurable entity types may be design-related but the project component should provide information on the derivation types that link the design-related entities to the "code" scope to capture the rationale of the assessment process. Additionally, the assessment model type should be defined as "predictive" in order to alert the stakeholders that the assessment values may not be fully reliable.

Coherence of values

Methods, functions and quality indicators indicate what to expect from the measurement and assessment values. They define the type these values comply to. As such, the MoCQA model may be inspected in order to detect any incoherence between the values and the operations performed on the basis of these values.

The value type, value range and scale provide information to ensure that the function and assessment model do not perform:

- inadmissible transformations on individual values (which are provided by [Zuse, 1997])
- inadmissible compositions of multiple values (which are explored in [Falcone, 2010])

For instance, a function relying on two base attributes which are assigned values comprised in an ordinal scale may not perform any arithmetic operation based on these values. Similarly, an assessment relying on a integer value comprised in a nominal scale may not perform any arithmetic operation on this value.

Coverage of interpretation rules

In order to be exploitable, the MoCQA model must not allow ambiguity on the interpretations associated to any quality indicator. The most basic method to ensure that no misinterpretation remains is to define a sufficient number of interpretation rules for each quality indicator.

Due to the nature of quality indicators, two types of situations may occur:

1. The quality indicator represents a finite set of values
2. The quality indicator represents a continuous series of values

The first case is identifiable thanks to the value type of the indicator. If the value type is defined as an enumeration, the MoCQA model should provide as many interpretation rules as the number of elements in the enumeration.

In the second case, the value range of the quality indicator may be used. In that case, the collection of all "range" attributes of the associated interpretation rules should:

1. Never overlap
2. Provide a total range that equals the quality indicator value range

As explained before, this verification does not guarantee that the interpretations themselves are correct (a validation that should be derived from the source material integrated to the MoCQA model, as well as the agreement of stakeholders) but that no value of the quality indicator is left void of meaning.

Robustness of the project component

The robustness of the project component may be defined as its capability to support an efficient root-cause analysis of the software project. Ensuring the robustness of the project component of MoCQA models consists in verifying that the definition of measurable entity types provide enough information to keep track of the possible sources of a detected flaw. The quality assessment metamodel provides several mechanisms adding constraints to measurable entity types (such, as the "subdivided in" associations, the source and target of the derivations types, etc.) to make their definition and identification more accurate. This information may and should be used to improve the traceability of defects. To an extent, the structural coherence of MoCQA models (checked during the design step) contributes to this robustness. For instance, since we know that a behaviour may not be corrected directly but requires some refactoring of the supporting code,

the quality assessment metamodel makes it mandatory to associate an artefact type to a behaviour type so that the development team is aware of the source of possible poor results.

However, the structure of MoCQA models do not guarantee that this traceability information is relevant or exploitable. Continuing on our example, the MoCQA model could provide an artefact type named "code" and associate it to the evaluated behaviour. In that case, the project component may not be considered robust since any root-cause analysis based on its definition would be directed to the obvious source code to correct any spotted defect. This information is not precise enough to help the quality assurance team perform the investigation.

In order to be robust, a MoCQA model should provide enough information to trace back a defect or a poor assessment result to its source, defining enough leads towards the probable cause of the defect. For instance, associating our behaviour type to the Java class `my.application.session` provides a clearer prospect of what could have gone wrong. Providing additional details such the fact that this precise Java class is the result of a derivation with source artefact types "class diagram X" and "sequence diagram Y" reinforces the robustness of the project component.

8.3.2 MoCQA-related indicators

MoCQA models are designed to be intrinsic artefacts of the software development process, like UML diagrams, software code, etc. As such, they qualify as measurable entities and may be assessed through measures, estimations and quality indicators. The remainder of this section defines and investigates several indicators that apply to MoCQA models and have been identified as relevant during the course of this research. These indicators may be derived from a MoCQA model in order to provide a better awareness of the overall performance of the planned quality assessment process.

Maximal definition of quality issue (MDef)

It may be interesting to be aware of the maximum accuracy the assessment performed could have reached, had the functions and assessment models been defined differently. MDef provides a quantitative estimation of the maximum sharpness a quality indicator could have provided and may be defined as,

$$MDef = \min s_1, s_2, ..., s_n$$

where $s_1, s_2, ..., s_n$ are the scales associated to each base attribute used by the assessment model of the evaluated quality indicator, converted as integer (i.e., nominal = 0 and absolute = 5).

Manipulations performed on the values may at best preserve the weakest scale among the input values, or weaken the overall scale. The weakest scale among the base attributes thus defines the strongest scale possible for a function or assessment model that relies on the values assigned to these base attributes.

MDef itself is associated to an ordinal scale, since it allows the ranking among its values but does not qualify as an interval scale.

Loss of definition for a quality indicator (LDef)

Based on the previous indicator, we may derive another property of MoCQA models that helps consider the relevance of the functions and assessment models we defined. The LDef indicator provides an estimate of the extent to which base measures have been weakened through the various transformations and compositions they underwent during the computation of a specific quality indicator. It may be defined as,

$$LDef = MDef - s_q i$$

where $s_q i$ is the scale of the evaluated quality indicator, converted as an integer. Provided that the content integrity has been correctly executed, the scale of a quality indicator should never be stronger than the scale of its inputs. Therefore, LDef produces a value comprised between 0 and 5 and indicates the measurement levels lost during the assessment process.

LDef itself is associated to an ordinal scale, since it allows the ranking among its values but does not qualify as an interval scale.

Criticality of entity types (NIssue)

The criticality of a type of measurable entity may be estimated by the NIssue indicator, which is defined as,

$$NIssue = N$$

where N is the number of quality issues that ultimately rely on the evaluated entity type. This indicator is associated to a ratio scale since it possesses a true zero point. This information may be used to direct the attention of the development team on a component that is crucial to the overall quality of the software project.

Range of quality issues (NEntitiy)

The range of a quality issue may be estimated by the NEntity indicator, which is defined as,

$$NEntity = N$$

where N is the number of measurable entity types that are used to satisfy the information need of the quality issue. This indicator is associated to a ratio scale since it possesses a true zero point. This information may help stakeholders realise how important and transversal a quality issue is.

Reliability of quality indicators (CI)

The reliability of quality indicators may be estimated through a very simple yet useful Confidence Index (CI) defined as,

$$CI = \min st_1, st_2, ..., st_n$$

where $st_1, st_2, ..., st_n$ are the status of every method, function or assessment model used to produce the indicator, converted as integer (i.e., experimental = 0, theoretically validated = 1, experimentally validated = 2, fully validated = 3). Relying on the concept that a chain (of operations in our case) is as strong as its weakest link, the CI indicators is defined by the least validated method, function or assessment model used to produce the quality indicator. CI is associated with an ordinal scale.

Product/Process orientation (Prod/Proc)

These two indicators may reveal the global purpose of the quality assessment process. The first indicator is defined as,

$$Prod = \frac{(NArt + NBehav)}{NEntityTypes}$$

where $NArt$ and $NBehav$ are respectively the number of artefact types and behaviour types *actually* measured in the process (for which an attribute is defined) and $NEntityTypes$ the total number of measurable entity types *actually* measured.

The second indicator is defined as,

$$Proc = \frac{NDeriv}{NEntityTypes}$$

where $NDeriv$ is the number of derivation types *actually* measured in the process (for which an attribute is defined) and $NEntityTypes$ the total number of measurable entity types *actually* measured.

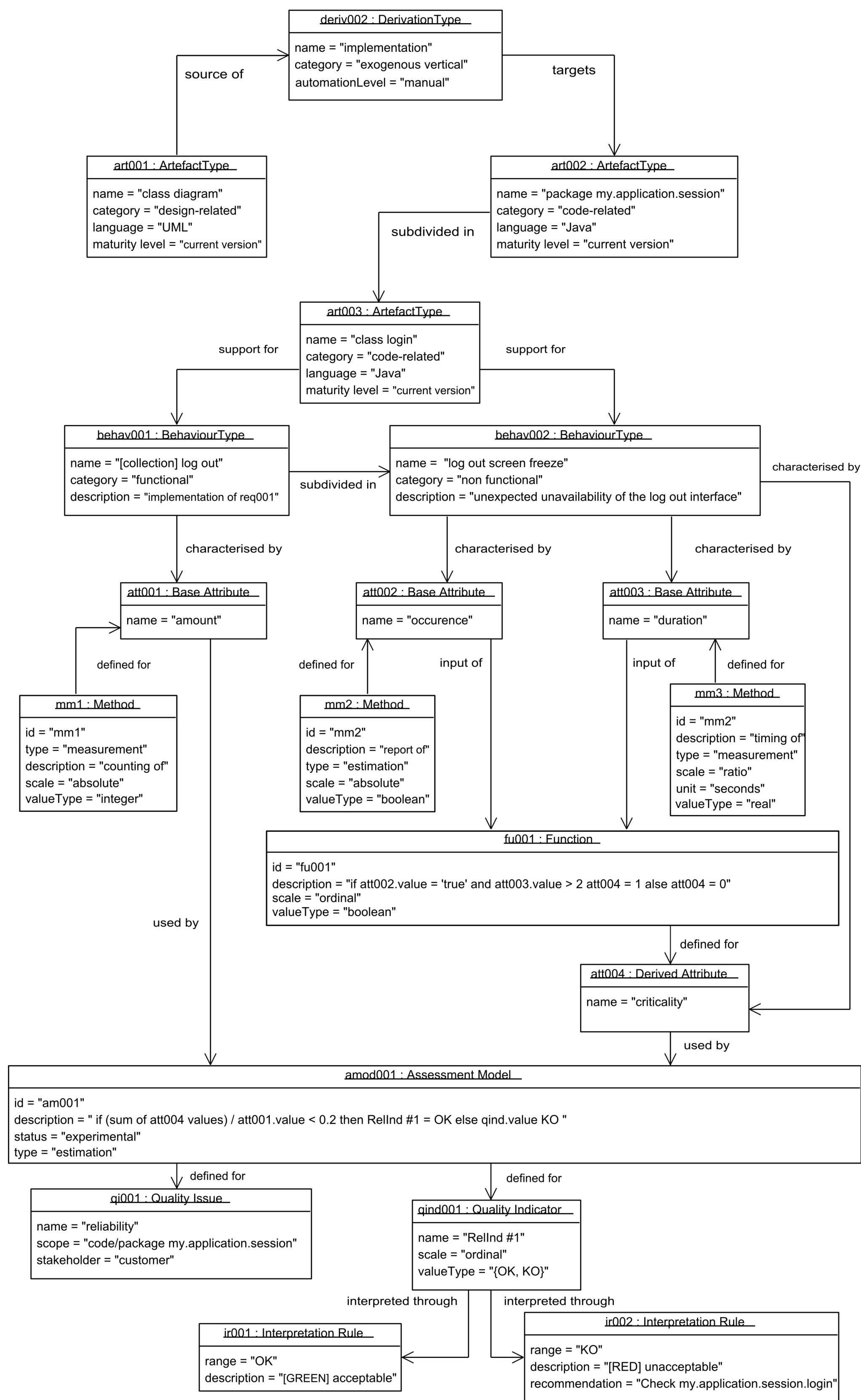

Figure 8.2: Example MoCQA model

8.3.3 Illustration

In order to illustrate the reviewing process of MoCQA models, let's consider the complete example designed in Chapter 5 and shown in Figure 8.2.

Regarding the content integrity, we have to verify the coherence of the scope and categories first. In our example, this aspect of the MoCQA model satisfies the integrity requirements. Indeed, although the main quality issue is associated to the `my.application.session` scope while the measurement is actually performed on the basis of behaviour types, the model provides the information required to trace back the measured entities to the scope of the quality issue. The rationale behind this assessment is therefore available to the stakeholders and indicates that the reliability is perceived as an external characteristics of package `my.application.session` (according to the classification of the ISO/IEC quality model).

Regarding the coherence of value types and the operations performed by function and assessment models, we find no violation of type or scale during the composition of the values. The values associated to base attributes are comprised in scales that are flexible (i.e., ratio and absolute) and the operations performed are mainly comparison between binary states. Therefore, the coherence of the values and their composition is verified.

Interpretation rules have been defined for each possible value of the quality indicator. The coverage is thus satisfying.

Regarding the robustness of the project component, although it models adequately the relationship between the scope of the main quality issue and the measured entities, it lacks details to support an efficient root-cause analysis. Indeed, the association between the assessed package and elements of design is specified but the design-related artefact type is not very useful in this context. Since the assessment performed focuses on an external characteristic, the occurrence of defect may be related to either environmental factors (e.g., a weak connection or slow hardware) or the implementation of behavioural elements. As such, the specification of an existing class diagram that was used to implement the assessed package is not as relevant. In case of poor measurement results, the improvement efforts will likely focus on the way the behaviour is implemented. The MoCQA model as it is does not offer traceability information regarding this concern. The documentation of which sequence or activity diagrams were used to implement the `my.application.session` package would have reinforce the robustness of the project component. With this information, the development team could have been able to check the quality of these elements of design, allowing them to determine if the conception of the system may explain the poor results.

Finally, we may apply some of the designed MoCQA indicators to the example. Definition-related indicators are the most relevant indicators to apply to the example since the MoCQA model is not complex and does not display a wealth of

quality issues. MDef provides a value of 4 while LDef provides a value of 2, which means that two measurement levels were lost, bringing back the final indicator to almost the weakest scale possible.

In consequence, improving the example MoCQA model would consist in providing more information regarding the behavioural diagrams used to implement the assessed package. Regarding the loss of definition of the quality assessment process, it may be explained by the fact that the assessment model uses the ratio computed on the basis of `att004` and `att001` to produce a binary indicator (OK, KO) while it could be used directly as a percentage of defect. This would provide a more accurate and fine-grained estimations of the reliability of the assessed package.

Chapter 9

Tool support

As we have seen in the previous chapters, the MoCQA methodology impacts the way quality assessment is performed. First, the approach introduces new artefacts (i.e., MoCQA models and formalised measurement plans) that have to be produced and maintained in an efficient way. Besides, the emphasis on the collaboration and communication between stakeholders implies that these artefacts have to be easily accessible and shared among the stakeholders. These aspects tend to increase the time and effort dedicated to quality assessment. However, the fact that the methodology is a full-fledged model-driven approach based on an explicit metamodel provides the opportunity to improve its usability (and therefore reduce the overhead regarding time and effort) through various types of tools.

This chapter provides an overview of the tool support that may be supplied in order to ensure the usability and effectiveness of the MoCQA framework. Section 9.1 enumerates the specific challenges raised by the methodology. Section 9.2 discusses how to take advantage of the existing tool support dedicated to model-driven development in the context of the MoCQA approach. Finally, Section 9.3 explores how a dedicated tool support could help approach the specific challenges of quality assessment modelling and describes tools that have been developed during the course of this research in order to improve the usability of the MoCQA framework.

9.1 Tool-related challenges of model-driven quality assessment

As explained in [Kent, 2002], *"tooling is essential to maximise the benefits of having models, and to minimise the effort required to maintain them"*. Beside

the basic model editing functionalities, the support expected by such tooling ranges from the possibility to ensure the well-formedness of the models to the possibility to work easily in a collaborative environment. Adequate tools may also improve the visualisation of complex models, ease their transformation into other types of models and enhance the overall exploitation of the information conveyed by the models. Model-driven quality assessment shares the same basic challenges as any model-driven approach and therefore requires the same type of tool support. However, model-driven quality assessment as envisioned in the MoCQA framework also raises some specific issues that may be dealt with through more specific tooling.

The main issue raised by the methodology, and specifically by MoCQA models, concerns the overall **scalability** of the approach. As we have seen in previous chapters, efficient quality assessment modelling requires more information than a traditional information product. This issue is caused, on the one hand, by the fact that MoCQA models intend to address the entire software project instead of a single entity. On the other hand, MoCQA models allow (or sometimes require) an important level of detail in order to specify crucial elements of the quality assessment process. As a consequence, MoCQA models may quickly become large and crowded with information. Mechanisms have to be provided in order to cope with this increased level of detail.

Another issue raised by the methodology lies in one of its core principles. The MoCQA framework intends to switch the focus of quantitative assessment from control mechanism to guiding mechanism. As such, the development team should be kept aware of the quality assessment performed on the resources they are acting on. In order to be supported, this principle requires the development team to have **access** to the MoCQA model that has to be centralised and easily available for consultation.

Finally, the integrative nature to the approach regarding quality models and existing measurement methods requires an easy and streamlined access to reference material. This calls for systematic ways to access this software quality knowledge in order to ease the integration of quality assessment methods from multiple sources.

9.2 Model-driven tools and MoCQA framework

Due to their explicit quality assessment metamodel, MoCQA models may be handled by any modelling environment with model-driven capabilities. Provided that a modelling tool allows the use of a metamodel as an abstract syntax, MoCQA models may be edited using this tool.

Due to its alignment with the Meta-Object Facility (MOF) architecture, the vast majority of tools supporting UML may be used to this end and produce

object diagrams that are compliant with the concepts of the quality assessment metamodel. This aspect of the methodology ensures its flexibility since the quality assurance team members may rely on the tools they already know (and use in their current workflow) in order to edit MoCQA models.

In order to improve the support of MoCQA models, the use of UML profiles may also provide a first step towards a domain-specific language, as explained in [Abouzahra et al., 2005]. Relying on stereotypes and tagged values, UML profiles allow the customisation of the UML formalism in order to provide a tailored concrete syntax for the approach. Several UML modelling tools used in the industry allow a thorough support of UML profiles (e.g., MagicDraw[1], Enterprise Architect[2], etc.).

Additionally, the fact that the MoCQA methodology is a full-fledged model-driven approach makes it possible to rely on model-driven specific techniques (and tools) that may be beneficial in the context of the quality assessment methodology.

9.2.1 Exploitation of model constraints

Defining constraints on models allows the definition of specific rules that the model has to comply to in order to be valid. The Object Constraint Language (OCL) [OMG, 2010] is a textual specification language that allows the definition of constraints on any model complying to the MOF architecture. The application of OCL constraints in the context of the MoCQA methodology is beneficial on several levels.

As part of a UML profile, OCL invariants (i.e., boolean OCL expressions) may be used to guarantee the validity of the designed MoCQA model regarding the structural and semantic validation described in Chapter 5 and 8, respectively. UML modelling tools with UML profile support generally offer OCL-related functionalities and may therefore be used in the context of the MoCQA methodology.

Since OCL allows the definition of constraints on both the well-formedness and the content of a specific MoCQA model, it also provides an opportunity to define a set of rules that apply to a specific software project. For instance, OCL constraints may be used to ensure that methods or functions linked to a given quality issue have to be associated with a specific scale or value type. Additional requirements may also be integrated in the MoCQA model through constraints. For instance, a constraint could be defined in order to guarantee that all child quality issues have the same referenced quality model as their parent quality issue.

Finally, OCL allows the definition of query expressions that may be useful in order to retrieve complex information from MoCQA models.

[1] http://www.nomagic.com/products/magicdraw/
[2] http://www.sparxsystems.com/products/index.html

9.2.2 Exploitation of model transformation languages

As explained in [Sendall and Kozaczynski, 2003], model transformations are the cornerstone of model-driven engineering. In consequence, several model transformation languages have been defined to formalise and automate these transformations (e.g., ATL [Jouault et al., 2006], QVT [Kurtev, 2008], etc.). As a model-driven approach, the MoCQA framework may rely on model transformation languages and their tool support in order to streamline the operationalisation step (see Chapter 7).

9.2.3 Co-evolution of models and collaborative modelling

Co-evolution is an intrinsic challenge of Software Evolution [Mens et al., 2005b] that is especially relevant in the context of model-driven development where the co-evolution between models and the code is essential. Contrary to model-driven development, the MoCQA framework alleviates this issue through its methodology. Indeed, the co-evolution between a MoCQA model and its concrete measurement plan is guaranteed by the successive steps within a quality assessment cycle. Similarly, the co-evolution between a MoCQA model and the data model used to store the measurement results is guaranteed by the specific data model described in Chapter 7.

However, the challenge of co-evolution resurfaces in the context of a distributed use of MoCQA models. In order to allow a better efficiency of the methodology, it may be beneficial to let quality assurance team members work on separate parts of the software project or quality issues. Among other benefits, this would ensure that the number of stakeholders a quality assurance team member has to include in the process is manageable. The most obvious solution to support this approach is to use several MoCQA models. However, this approach raises some issues. In the context of the MoCQA approach, working on different quality issues does not automatically implies working on separate parts of the project (and conversely). These hypothetical separate models would therefore probably possess common subsets of elements. In this case, the co-evolution between the models would become a complex issue.

In order to address this issue, collaborative modelling (i.e., several separate individuals actively contributing to the creation of a single model [Rittgen, 2009]) may prove useful. This approach would guarantee that a single central MoCQA model is maintained for the entire software project (and for any one to consult). Collaborative modelling would in the meantime allow separate teams or individuals to work on more focused concerns, without the risk of incoherence. Several efforts have been carried out to address the challenges of collaborative editing (e.g., [Rittgen, 2008] or [Koshima et al., 2011]) and should be investigated in the context of the MoCQA framework.

9.3 Dedicated and integrated tool support

Although existing model-driven tools provide the quality assurance team with suitable support for applying the MoCQA methodology, they remain general-purpose and do not offer solutions that are tailored to the specific challenges raised by the methodology. In order to address these specific challenges, a dedicated set of tools is still required. During the course of this research, several tool-related efforts have been carried out in order to improve the efficiency of specific tasks of the methodology. As such, they provide the core components for a future dedicated and integrated tool support that would fully leverage the potential of the framework.

9.3.1 XML-based Operational Customised Quality Assessment Model

As explained in Chapter 7, the operationalisation of a MoCQA model is an exogenous and vertical model transformation, regardless of the formalism used to express the measurement plan. XOCQAM is a formalism that has been defined to allow this model transformation. XOCQAM is an XML-based language defined by its own metamodel (i.e., an XML schema). The goal of using an XML-based formalism is twofold. First, the format is sufficiently structured to provide a model that may easily be manipulated (e.g., edition, modification, importation/-exportation, filtering, etc.) and stored. Besides, as a well-known mark-up language, the XML formalism provides a good textual notation that complements the graphical notation used by the quality assurance team to express MoCQA models.

Additionally, XML-based languages are efficiently supported by several editors (e.g., validation of XPATH constraints, auto-completion based on XML schemas, etc.). XOCQAM therefore provides an additional layer to support a domain-specific language and improves the expressiveness and usability of MoCQA models, as well as their reusability.

A typical XOCQAM document is divided in three components (i.e., project, measurement and assessment) in the same way the MoCQA models are defined. Whereas the project component includes the same constructs as the project package of the quality assessment metamodel, the measurement component of XOCQAM only allows attributes constructs and the assessment component only includes quality issues. This is due to the fact that XOCQAM takes advantage of the hierarchical nature of XML in order to keep XOCQAM documents clean.

An XOCQAM element possesses the same attributes as its MoCQA counterpart and integrates the information on mandatory attributes. In order to facilitate the identification of elements, each XOCQAM element is assigned an “id” that serves the purpose of primary key.

Two distinct mechanisms to support the modelling of associations between MoCQA constructs are provided:

- Reference to associated elements
- Inclusion of child elements

The reference mechanism consists in adding child elements that possess only a “ref” attribute within the parent element. This “ref” attribute acts as a foreign key. The inclusion mechanism consists in adding the referenced element directly in the parent element.

XOCQAM elements at the centre of the operationalisation process (i.e., assessment models, functions, methods and measurable entity types) also possess attributes provided to record the additional information required for the operationalisation (see Section 7.3).

Although XOCQAM embeds several structural constraints validation (i.e., mandatory attributes, validation of the cardinality of the source of an element), it does not check constraints on associations in both ways. For instance, the XOCQAM schema ensures that an assessment model produces at least one quality indicator but no constraint is defined to verify that one specific quality indicator is associated to at most one assessment model. This design choice intents to limit the verbosity of the textual notation. It does not constitute a major hindrance since the use of XOCQAM normally follows the structural validation of the design step.

9.3.2 MoCQA Utilities on the Go (MUG)

Objectives

MUG is a graphical MoCQA modelling tool that intends to support the elicitation of quality requirements and their translation into MoCQA models. It intends to help ensure the scalability of the MoCQA approach and to provide an improved usability through specific navigation mechanisms. As such, it aims to be portable, focused on user-friendly and easy navigation within large MoCQA models.

Overview

Written in ActionScript 3.0[3] and relying on the open-source Adobe Flex framework[4], MUG provides a graphical environment to design MoCQA models. The choice of ActionScript as programming language is due to its high portability, especially for web-based applications and mobile devices (the latter being especially promising in the context of the elicitation of the quality requirements and offering additional flexibility to the process).

[3]http://www.adobe.com/devnet/actionscript.html
[4]http://www.adobe.com/be_fr/products/flex.html

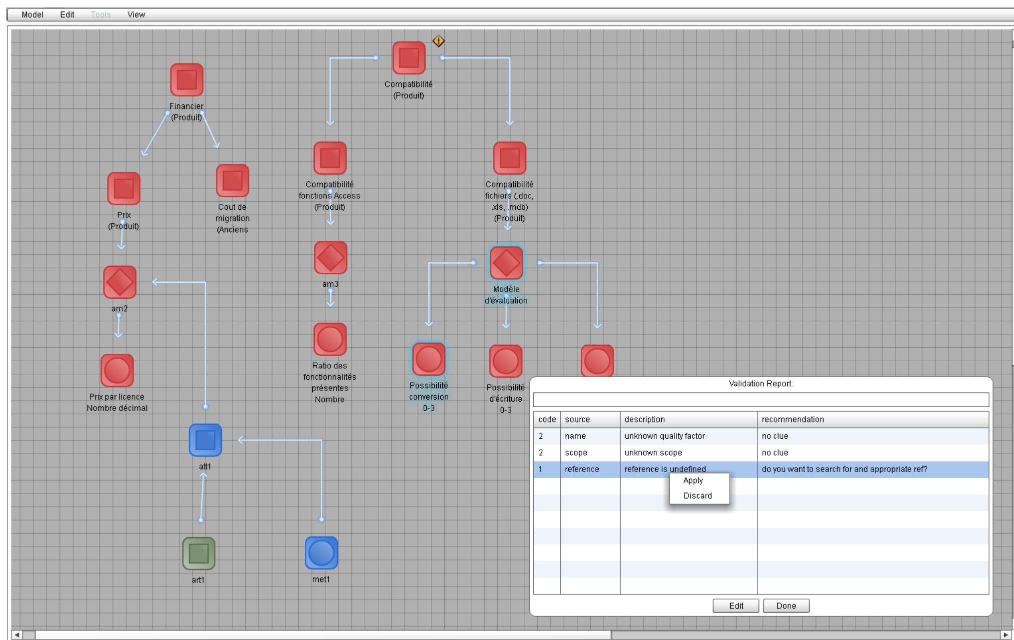

Figure 9.1: MUG user interface

As shown in Figure 9.1, MUG relies on an experimental graphical notation (based on the recommendations of [Moody, 2009])that intends to improve the usability of MoCQA models in a concrete context, that is, a more operational notation. In order to improve the legibility of large MoCQA models, the MUG graphical notation limits the displayed information to a subset of essential concepts and attributes. It therefore focuses on a broader (instead of a detailed) view of MoCQA models.

In order to provide the detailed view, the complete information contained in the model is available through different widgets built in the MUG tool. Implementing the heuristic stating that GUI designers should *"include in the displays only information needed by the user at a given time"* [Gerhardt-Powals, 1996], the complete information on attributes is accessible as an "infotip" or via dedicated forms to edit the attributes.

MUG provides several functionalities designed to increase its usability. Among others, it provides a cross-attributes search engine, a hinting system based on a internal repository of standard quality frameworks, the ability to navigate links dynamically and the ability to create multiple associations at once.

It relies on both a dedicated XML-based file format and also offers the ability to export file in XOCQAM format (although the edition of the specific metadata). It also provides the ability to import MoCQA models into a parent model in order to improve the reusability of the approach. Additionally, the MUG tool implements structural and some level of semantic validation of MoCQA models.

Finally, MUG is designed primarily as a web application but may be ported easily on any device (e.g., desktop computer running any OS, tablet devices, etc.) due to its underlying Flash technology.

9.3.3 OCQAM editor

Objectives

The OCQAM editor is an Eclipse plug-in providing a hierarchical MoCQA modelling tool. It intends to support the consultation of quality-assessment-related information by the development team during the course of the development. As such, it aims to improve the integration between the quality assessment life-cycle and the development/maintenance life-cycles.

Overview

Relying on the Eclipse Modeling Framework (EMF)[5], the OCQAM editor provides a hierarchical textual MoCQA model editor. As part of the Eclipse IDE[6], the editor allows developers to consult a MoCQA model designed for the current project conveniently and at any time.

Thanks to the editor, developers may quickly consult the quality issues and measures planned for a specific artefact they are working on. The editor allows OCL queries to be performed on the MoCQA model in order to consult information in a more refined way.

The file format used to store the models is a variant of XOCQAM, adapted to the requirements of EMF-based models. The interaction between MUG and the OCQAM editor is thus guaranteed.

9.3.4 QuaTALOG

Objectives

The knowledge required to deploy the MoCQA framework includes not only environmental information but also some background (or general) information on quality assessment and measurement methods available and/or imposed to the analyst. Given all the information collected through various stakeholders, the aim of the MoCQA methodology is to determine valid and adequate methods to assess and monitor the goals (with respect to the available resources for a given project) and integrate them into a MoCQA model. However, the task is hindered by the fact that information collected from stakeholders may not be structured. The analyst may also not have a comprehensive knowledge of the different available quality models and their differences and commonalities.

[5]http://www.eclipse.org/modeling/emf/
[6]http://www.eclipse.org/

Besides, the analyst may be confronted to additional constraints and challenges while designing a MoCQA model. For instance, an existing quality model could suggest the use of a given set of metrics designed to evaluate the source code (e.g., suggested metrics for maintainability in ISO/IEC 9126) whilst the analyst has to face the evaluation of the same quality characteristic (i.e., maintainability) for the design, therefore requiring an alternative measurement method targeting the same quality characteristic.

Another challenge for the analyst who designs a MoCQA model would be the requirement to adapt a given quality assessment/improvement method during a certification process. In this case, the analyst would benefit from an easy way to compare the measurement methods of the current quality assessment process with the ones specified in the norms, in order to check if upgrades are required or if the current methods already comply to the standard. The same applies in case of a change from a set of given standards A to a second set B.

Introduced in [Vanderose and Habra, 2011] the QuaTALOG tool aims to provide functionalities to maintain an up-to-date, structured and extensive catalogue of available quality assessments methods (e.g., quality models, measurement methods, etc.). The other goal of the tool is to provide a user-friendly access to this knowledge.

Overview

Relying on the open-source Apache Struts2 framework[7], QuaTALOG provides the following functionalities:

1. A repository of data that is structured to be easily integrated into a MoCQA model;
2. A user interface to edit and consult the content of the online database;
3. A keyword-based and concept-based search engine;
4. A support for the distant interrogation of the database by third-party tools and most notably MoCQA model editors.

As shown in Figure 9.2, QuaTALOG takes the form of a web-based knowledge base with a standard (and thus intuitive) interface that also provides standard browsing web services. The knowledge base allows the quick navigation between (even remotely) related concepts. It offers a way to check out the options available to the user (i.e., the MoCQA analyst or any quality assurance manager) for his specific needs. The level of validity of a specific assessment model (i.e., number of existing empirical validations, number of theoretical validations) is also specified in order to provide additional guarantees (or an increased cautiousness) to the analyst. The tool has been specifically optimised to support the analyst during the design of a MoCQA model. It has also been designed to be available to

[7] http://struts.apache.org/2.x/

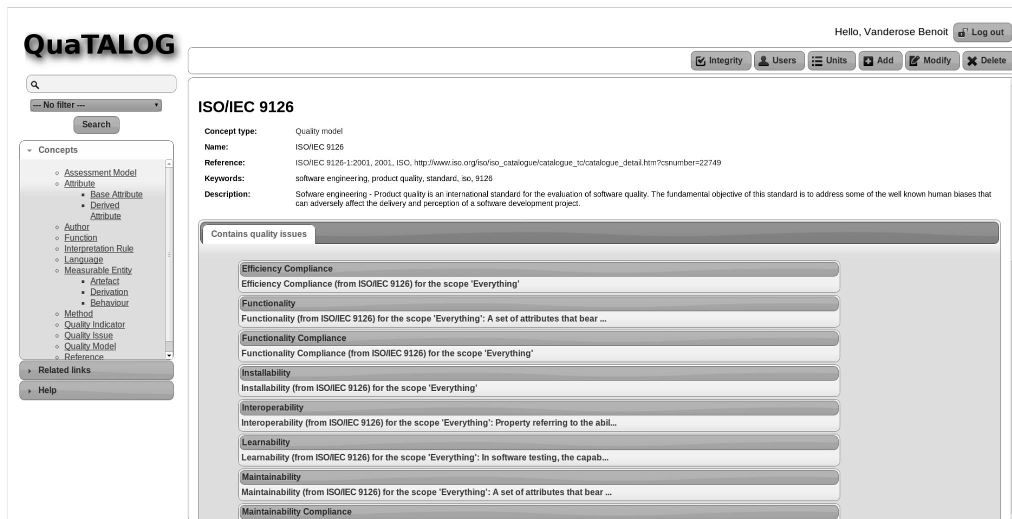

Figure 9.2: QuaTALOG user interface

anyone willing to consult and compare quality assessment methods, or willing to contribute to the catalogue.

Architecture

From an architectural point of view, the key aspect of QuaTALOG resides in the way the conceptual schema of the repository has been designed. Many architectures would have been valid to create a catalogue of quality assessment techniques. However, the compatibility with the quality assessment metamodel is essential. Therefore, the repository conceptual schema has been designed essentially as a series of transformations based on the quality metamodel. Those transformations were not always horizontal (i.e., conserving the same level of detail [Mens and Gorp, 2006]).

The process of transformation from metamodel to repository requires to adapt several elements. First, attributes from the metamodel have to be refined. For instance, the *reference* attribute of the *quality issue* concept had to be modified to support the storage of BibteX-formatted references).

Also, the cardinalities of some associations in the metamodel are designed to limit and structure MoCQA models (e.g., in a MoCQA model, a quality issue has at most one parent representing a more general goal while in the database, we want to record all possible parents of a given quality issue in order to provide the user with relevant alternatives). Finally, usual notions have to be aligned with MoCQA concepts. For instance, the notion of quality model does not appear explicitly in the quality assessment metamodel but had to be added in the QuaTALOG database schema. The Software Quality ontology was used as a support for these additions. Besides, the difference between quality characteristics

and quality issues had to be taken into account when linking the notion of quality model to the quality issue (i.e., a quality issue is a quality characteristic with a scope and stakeholder while traditional quality models set the scope of all quality characteristics at once).

This process of transformation ensures that the data is stored in a structured way in order to be compared, aligned and integrated in a MoCQA model. Based on previous case studies [Vanderose et al., 2010], the availability of such a tool could decrease the time of design for a MoCQA model by up to 40-60%, provided that the catalogue is correctly populated. Indeed, feedback from the previous case studies always points out that more than 50% of the MoCQA model design time was devoted to finding adequate techniques. However, in these cases, the analysts were students with little or no experience in software quality and an experienced analyst should find adequate methods faster, although not systematically look for better alternatives to the methods she knows.

9.3.5 Towards and integrated tool support

So far, the tooling developed to support the framework is comprised of distinct tools focusing on specific aspects of the MoCQA methodology. The improvement and integration of these tools would directly benefit the approach and provide a more thorough support for the framework.

Interactions between tools

In order to provide a better coverage of the methodology, the existing tools should be integrated in a tighter workflow. Since the Eclipse-based OCQAM editor and the MUG tool already exchange compatible data, this effort should focus on the integration of the QuaTALOG platform. By refining the existing support for web services provided by QuaTALOG and allowing the MoCQA model editors to consult and integrate knowledge retrieved through these web services, the workflow would gain in efficiency.

The MoCQA Toolkit

As explained in [Kelly, 2004], two options are available to provide a tailored tool support for a Domain-Specific Modelling (DSM) approach: metaCASE tools and DSM coding frameworks.

MetaCASE tools (e.g., MetaEdit+[8]) are dedicated to the generation of CASE tools on the basis of a provided metamodel. DSM coding frameworks (e.g. the Eclipse Modeling Framework used in conjunction with the GMF tooling project[9]), on the other hand, require more intensive software development in order to obtain

[8] http://www.metacase.com/MetaEdit.html

[9] http://eclipse.org/gmf-tooling/

the final tool. Unsurprisingly, metaCASE tools provide results faster and easier. Additionally, using powerful metaCASE tools such as MetaDONE [Englebert and Heymans, 2007] provide much flexibility in order to acquire a fine-tuned support.

However, the ongoing and future development of the MoCQA specific tooling not only requires DSM support but should be integrated more closely with the software development environment. In that context, relying on the Eclipse environment appears as a more suitable option. First, the Eclipse environment is primarily an Integrated Development Environment. The integration of the MoCQA-specific tool support into this environment is thus a suitable way to guarantee the closer integration between the quality assessment life-cycle and the development life-cycle. Besides, the Eclipse environment provides many projects focusing on model-driven support[10]. Relying on the Eclipse environment is thus a way to easily take advantage of the techniques described above (i.e., OCL, automated model transformations, etc.).

As such, the current OCQAM editor should be used as the central component for a complete MoCQA Toolkit. The following components would help cover all the essential aspects of the MoCQA methodology in an integrated way.

A **quality dashboard** component would use the active MoCQA model loaded thanks to the OCQAM editor. Its goal would be to present a view of the various quality indicators associated with the quality issues. It would use the interpretation rules of the MoCQA model to provide a comprehensive view of the issues. Among other functionalities, the MoCQA quality dashboard would allow filtering the information by stakeholders, scope, artefacts, etc. The quality dashboard would also be reconfigured in real time if the active MoCQA model is edited in the OCQAM editor.

A **project outliner** component would help identify the actual resources that are prone to be assessed. The aim of such a component is to ensure the awareness of the developer about the existence of a quality assessment strategy (i.e., to be aware that the piece of code she is editing is in fact measured to guarantee a given quality goal). It mainly relies on the project-related component of the MoCQA model. Each entity type defined in this component would provide a tag (implemented by an Eclipse marker) to the project outliner. These tags would be used primarily to point out the specific resources that have to be measured. They would also be used to bind the actual resources with the active MoCQA model, allowing a view on the quality dashboard focused on this specific resource, from within the project explorer of Eclipse.

Finally, in order to prepare the automation of the process, a **measurement manager** component should be developed. This tool would constitute a configuration panel that helps bind the measurement/estimation methods described

[10]http://www.eclipse.org/modeling/mdt/

in the MoCQA model with actual measurement procedures provided by external tools. It would also define all the relevant data for the management of the measurement plan (e.g., associating a measurement procedure with a defined tag, defining the frequency of the measurement, etc.).

Part III

Validation of the approach

Chapter 10

Validation process

Part II described a theoretical approach that is designed to help integrate quality assessment into the development life-cycle in a more effective way. The MoCQA framework implements this approach and provides a practical assessment methodology designed to leverage the expected benefits of the theoretical approach. This chapter describes the evaluation process we applied in order to ensure that the framework is usable and demonstrates some of the desired benefits elicited in Chapter 4.

10.1 Research questions

In order to evaluate the potential of the MoCQA framework, we first identified a series of research questions to guide our effort. Each of these questions may be associated with one of the two main topics identified below.

The first topic of our validation process pertains to the usability of the framework. Provided that the framework under review relies on a quality assessment methodology that introduces specific notions and activities, the first research question to investigate its ability to be used in practice. The question mainly addresses the fact that it is possible to apply the quality assessment methodology, step by step, without any major hindrance or overwhelming increase in time and effort for the development or quality assurance team.

The usability of the framework also relates to the overall acceptance of the framework by the stakeholders. In other words, we have to check that no major reluctance regarding the participative nature of the approach arises during the deployment.

Finally, the first question also covers the fact that all stakeholders understand the methodology and that MoCQA models are adequate to communicate the

information between the involved parties.
These concerns may be formalised as the following research questions:

[**RQ1**] Is the MoCQA framework usable?

[**RQ1a**] Is the model-driven quality assessment methodology defined by the framework applicable in practice without negative impacts on the rest of the development process?

[**RQ1b**] Is the MoCQA framework accepted and adopted by all involved stakeholders?

[**RQ1c**] Are MoCQA models apt to model the necessary quality assessment information and support the communication of this information between stakeholders?

The second main topic of the validation process relates to the effectiveness of the framework. Indeed, the implicit claim of the MoCQA framework is that its application helps rectify some of the shortcomings pertaining to Software Quality as a field. More specifically, the claim is that the core theoretical notions introduced in the framework are expected to provide some benefits to the user of the framework and the overall development team, as explained in Chapter 4. The second main research question is therefore to determine if the framework effectively leverage the expected benefits.

The fundamental advantageous aspect of the framework is the fact that MoCQA models and the methodology used to exploit them helps adapt to a specific context and model accurately the quality requirements for a given project.

The second expected advantage of the framework is that it allows a targeted assessment. Targeted assessment means that each evaluation effort is carried out to fulfil a specific information need, therefore avoiding an unproductive measurement plan. The iterative and incremental nature of the methodology ensures that new quality requirements can be treated in the next quality assessment cycle, which prevent the risk of forgetting any important quality goal.

The iterative methodology also allows the refinement of quality indicators and their evaluation methods as the development progresses. This constitutes the third expected benefit: the framework should allow a better integration of quality assessment, from the earliest stages of the development.

Another advantage of the model-driven quality assessment is its supposed ability to improve the self-awareness of the process. The self-awareness denotes the ability to rely on the MoCQA models to detect and rectify the quality assessment process. In turn, this self-awareness of quality assessment helps converge towards the elicited quality goals.

Finally, the use of explicit and integrated quality assessment modelling is supposed to provide a better support for the analysis of the current quality level and the identification of corrective actions on the project.

Those concerns result in the following research questions:

[**RQ2**] Is the MoCQA framework effective?

[**RQ2a**] Does explicit and integrated quality assessment modelling succeed in accurately modelling the specific quality requirements for a given context?

[**RQ2b**] Does the quality assessment methodology helps provide a targeted assessment that meets the specific quality requirements?

[**RQ2c**] Does the iterative use of MoCQA models help plan and adjust the quality assessment process throughout the software life-cycle, from early stages to maintenance and evolution?

[**RQ2d**] Do MoCQA models help detect the flaws in the quality assessment process that is performed?

[**RQ2e**] Do MoCQA models help identify the corrective action that have to be performed in order to improve the level of satisfaction of the quality goals?

10.2 Challenges

The nature of the research work presented in this dissertation raises some issues regarding the validation process. The introduction of a new type of methodology can only gain acceptance throughout time and repeated opportunities to apply it. Additionally, the validation of a quality assessment methodology raises even more challenges: Software quality is such a transversal topic that a complete validation of the approach would require a huge amount of time and effort that extends the scope of this dissertation. Similarly, although the need for empirical evidence in software engineering researches is stressed by many authors [Juristo and Moreno, 2001 Wohlin et al., 2000], the acquisition of statistically significant empirical data represents a complex challenge in our context.

In order to illustrate the extent of the effort required to obtain a satisfactory level of validation, let's consider an ideal validation protocol to answer the research questions identified in the previous section.

This ideal validation protocol would have to be applied in an industrial context. In order to provide an efficient way to evaluate the impact of the MoCQA framework, the context should allow two separate software development life-cycles to be performed on the same set of requirements. The first development life-cycle (DLC1) would apply traditional quality assessment techniques. The other development life-cycle (DLC2) would integrate the MoCQA framework. Additionally, the lifespan of the validation process should encompass the entire development

life-cycle, from the early stage of requirements engineering to the maintenance and evolution processes.

From that point on, the ideal validation protocol would consist in showing that the MoCQA framework allowed stakeholders from DLC2 to:

1. monitor and guide the development process from the earliest stage of development;
2. define a context-specific model that meets the expectation of the stakeholders;
3. communicate between stakeholders (from developers to managers);
4. detect the flaws or misuse of measures during the quality assessment process;
5. guide the maintenance and evolution process for their project.

The validation should demonstrate that these achievements were met without any unacceptable overhead, any increase in cost or any delay, in comparison to DLC1. Finally, the validation protocol should demonstrate that the overall satisfaction of the stakeholders of DLC2 is higher.

Although finding such a suitable context is unrealistic, this ideal validation protocol provide us with valuable insights on how to perform a manageable evaluation of our approach. When looked at transversally, the validation described above reveals two separate needs:

- Show that the framework is sustainable in a professional context
- Show that the framework provides better results

Therefore, we can divide the overall process into more manageable case studies, some of them focusing solely on the results obtained with the framework, others focusing on the application on the field. Our validation approach thus focuses on providing hints that the MoCQA framework is adequate by fragmenting the validation into several case studies.

In order to achieve this validation process, the research questions presented in the previous section have been broken down into several criteria that the approach should meet in order to satisfy the validity requirements. Each of these criteria encompasses some properties the approach should demonstrate in order to satisfy to specific aspects of the research questions.

Those criteria have been selected in order to allow some level of modularity towards our idealised process. The validation process therefore becomes context-independent. While some criteria cannot be assessed without a practical application, some other may be addressed through isolated theoretical case studies.

The identified criteria are the following: expressiveness, integrability, adaptability, exploitability and applicability. The remainder of this section details each of them (i.e., what properties the criterion covers, how it contributes to answering our research questions and how it allows us to keep our validation process manageable).

Expressiveness

Scope: Expressiveness covers the ability to express and integrate various quality models (or parts of them) as a hierarchy of quality issues into a MoCQA model. It also addresses the ability to describe measurement methods and link them to the quality issues hierarchy. Finally, it also includes the ability to express the right context through the project package of the quality assessment metamodel as well as the ability to express easily and accurately the measurable entities.
Target: This criterion contributes to answering RQ1c, RQ2a and RQ2b.
Comments: In order to assess the criterion, we may evaluate the integration of various quality models by comparing the quality assessment metamodel to explicited quality metamodels. The measurement package does not require much validation since it already builds on [ISO/IEC, 2007a]. The project package can be assessed thanks to targeted studies of customised measures and how much the constructs of the quality assessment metamodel help express them in a adequate way.

Integrability

Scope: Integrability covers the ability of the framework to be used in conjunction with other quality approaches and to fit the specific development or maintenance process defined in the environment.
Target: This criterion contributes to answering RQ1a, RQ1b and RQ2c.
Comments: This criterion allows us to provide some hints at the usefulness of the approach in early stages of the development from a theoretical point of view without requiring an empirical study. Case studies on the field can therefore focus on any stage of the development.

Adaptability

Scope: The adaptability of the framework represents its ability to provide a quality assessment that is tailored to a specific project (i.e., relying solely on the available resources, defining quality issues that are useful for their target stakeholders and evaluation methods that fit the expectations).
Target: This criterion contributes to answering RQ1c, RQ2a and RQ2b.
Comments: This criterion can be assessed through any category of case study.

Exploitability

Scope: Exploitability covers the ability of MoCQA models to provide quality profiles that are fit as a basis of the decision-making regarding the development life-cycle. It also covers the fact that the quality profile (and therefore, MoCQA models) can be used successfully to communicate among stakeholders. Finally,

exploitability encompasses the ability of MoCQA models to support the detection of flaws in the quality assessment process.
Target: This criterion contributes to answering RQ1b, RQ1c, RQ2d and RQ2e.
Comments: Separating the exploitability from the pure applicability of the approach allows us to perform theoretical case studies that focus on the earlier stage and show the potential of the framework without requiring an actual empirical study.

Applicability

Scope: The applicability of the framework specifically relates to the environment and the stakeholders. It encompasses the ability to be applied in a concrete context. It covers the effort and time the application of the quality assessment methodology requires and its ability to generate acceptance among the various stakeholders
Target: This criterion contributes to answering RQ1a, RQ1b, RQ2c and RQ2e.
Comments: Although the previous criteria all relate to some level of applicability, actual case studies using the MoCQA framework in an actual professional environment are required in order to validate the approach globally. However, the entire software life-cycle is difficult to assess. In order to validate the global usability, a new project (and therefore a close collaboration) is required. Defining an applicability criterion helps us focus on a subset of the development life-cycle (e.g., the maintenance phase) and assess how the actual stakeholders react to the framework in a professional context for this subset. Coupled with theoretical studies assessing the integrability and exploitability, a practical study on the applicability provides enough hints regarding the global usability of the framework. Another point covered by the criterion is the sustainability. Many of the 9 principles listed in Chapter 4 are already known for their benefits but also increase the effort and time needed to apply them. The key aspect to asses the applicability is therefore to check if the overhead induced by the application of these principles is balanced by the dedicated support provided by the framework.

The following chapters describe case studies designed to assess the criteria identified above:

- Chapter 11 investigates the expressiveness and integrability of the approach from a theoretical point of view.
- Chapter 12 provides a theoretical case study that explores integrability and exploitability at an early stage of development.
- Chapter 13 collects case studies, both theoretical and empirical, that focus on exploitability, adaptability and expressiveness.

- Chapter 14 describes a practical case study performed in a professional environment that illustrates the adaptability and applicability of the MoCQA framework.
- Chapter 15 reports a one-year long practical case study performed in a professional environment and mainly addressing the applicability of the MoCQA framework. Additionally, the case study also investigates expressiveness, adaptability, and exploitability.

Chapter 11

Operationalisation of quality models

This chapter describes a theoretical case study that illustrates the integrability and the expressiveness of the MoCQA framework. The case study exemplifies the first acquisition method (i.e., operationalisation of quality models) described in Chapter 6.

11.1 Objectives

The aim of this case study is twofold. First, it intends to illustrate the fact that the quality assessment metamodel is suitable to express different quality models and integrate them into a MoCQA model. Secondly, it exemplifies how quality models may be used to support the acquisition step and explore the feasibility of this acquisition method.

The case study focuses on two representative quality models. The first is the ISO/IEC quality model. This quality model is representative of a generic quality model, and also summarises most of the concepts of previous quality models (i.e., McCall's, Boehm's, etc.). The second quality model studied in this chapter is a specialised quality model focused on the availability of documentation in an open-source context. It is representative of ad hoc and specialised quality models constructed from scratch.

For both models, the study consists in a short overview of the compatibility between the MoCQA quality assessment metamodel and an explicitly derived quality metamodel (i.e., an illustration that all concepts of the quality model studied may be included in a MoCQA model). This overview is followed by an attempt to perform a generic acquisition step (i.e., an instantiation of the quality assessment metamodel without any hypothesis on the scope of stakeholders

involved in the process) based on the studied quality model. In other words, this instantiation attempts to recreate an entire instance of the quality model studied in terms of MoCQA concepts and investigates its potential to support the acquisition step.

Note that, although it is not covered in this study, an example of actual use of McCall's model within a MoCQA model can be found in Chapters 13.

11.2 ISO/IEC 9621 quality model

11.2.1 Overview

As explained in Chapter 1, the ISO/IEC 9126 quality model (referred to as ISOQM in the remainder of this section), is a quality model that addresses all the aspects of the software product. It is decomposed in six quality characteristics that are further refined in sub-characteristics and add characterisation of the in use quality structured in four characteristics. For each sub-characteristic, measurable attributes are defined, as well as metrics dedicated to their evaluation.

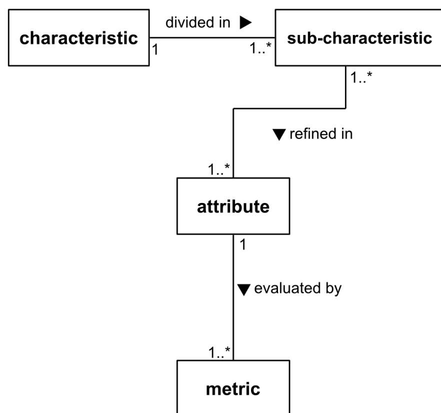

Figure 11.1: ISOQM explicited metamodel

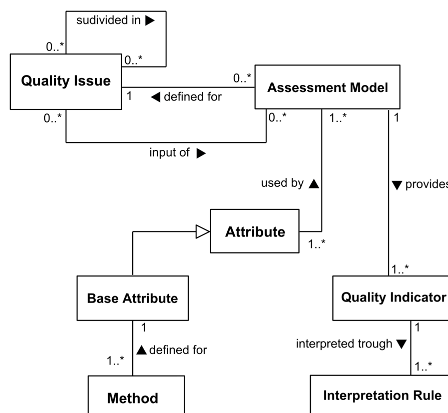

Figure 11.2: Related quality assessment metamodel concepts

As shown in Figure 11.1, expressing an explicit metamodel for the ISOQM is straightforward. The main concepts of the quality metamodel are characteristics that may be refined in sub-characteristics associated to attributes, themselves linked to metrics. The concepts illustrated in Figure 11.2 are concepts from the MoCQA quality assessment metamodel that may be aligned with the concepts of the ISOQM. As we may see, quality issues may be used to represent characteristics and sub-characteristics. Attributes have the same function in the two metamodels (although the quality assessment metamodel distinguishes base from derived attributes). Finally, the method concept of the quality assessment metamodel encompasses the notion of metrics, with a more generic scope. The

comparison of the two metamodels shows that the ISOQM metamodel requires a subset of the concepts of the MoCQA quality assessment metamodel. Indeed, the MoCQA quality assessment metamodel provides the additional quality indicator and interpretation rule concepts that are not explicitly required in the ISOQM metamodel. In consequence, the quality assessment metamodel is theoretically suitable to produce an ISOQM instance.

11.2.2 Instantiation

The most straightforward process to represent the ISOQM in a MoCQA model would be to define 6 quality issues for each external and internal characteristics, plus 4 others for each in use quality characteristic. However, we have to duplicate the first 6 quality issues in order to differentiate external and internal characteristics. The minimal number of quality issues is thus 16. Defining the scope of those quality issues is also not trivial without actual requirements from the stakeholders. If we define the scope as 'the software product', the resulting MoCQA model will not provide a really meaningful assessment due to the extent of this scope. If we decide that a smaller scope has to be defined (design, code, etc.), this will induce a duplication of the quality issues. There is no hint about any computation involving the subcharacteristics and, consequently, all the characteristic-inspired quality issues will be composed (not aggregated). Regarding the assessment models, no indication about how to link the various metrics assessing the attributes and the subcharacteristics except a 'participate to' kind of relationship is provided. Therefore, no quality indicators will be part of our MoCQA model.

The instantiation of the project component raises even more questions. The definition of software product used in ISOQM is very large and would require the definition of many artefacts in order for the MoCQA model to be relevant.

Finally, the instantiation of the measurement component also reveals a need for more specific goals. As a matter of fact, the metrics described in [ISO/IEC, 2001b;c;d] are described in very generic terms and need a certain amount of specialisation (they are more like templates for metrics). For instance, the metric *response time* defined as an efficiency metric has for defined purpose *"what is the estimated time to complete a specified task?"*. This metric could be applied to many different in use behaviours and taking all these possibilities into account would increase the number of measurable entities.

The instantiation is thus not impossible practically but requires an actual environment in order to answer some questions and provide the necessary constraints to create a ISOQM-based MoCQA model.

11.2.3 Results

The previous section shows that it is possible to generate a MoCQA model that translates the ISOQM, provided that some constraints are verified. The conclusion that may be drawn from this attempt is that a complete instantiation of the ISOQM would lead to a huge MoCQA model. One of the defining aspects of MoCQA models is the increased level of detail required in the definition of the measurable entities. The instantiation of a MoCQA counterpart of the ISOQM would require a systematic description of all the aspects of the 'software product' (all elements involved in design, requirements, documentation, code, etc.) and would lead to the duplication of quality issues of varying scopes (design, code, etc.). Indeed, the software product as defined in ISOQM (see Chapter 3) represents a collection of many types of behaviours and artefacts in the context of MoCQA.

The observation is also valid for the metrics associated to the ISOQM. They are defined in a very general way that would require a lot of duplication (time of response (base attribute) of each in use behaviour).

The question of how to reduce the size of this instantiation leads to two alternatives. The first one consists in suppressing some branches and focusing on the operationalisation of a single branch while keeping the 'software product' approach (e.g., functionality (quality issue) of the software product (scope)). The second requires the specialisation of measurable entities (e.g., the complete structure of internal quality characteristics (quality issues) for the requirement-related artefacts (scope)). In any case, these observations tend to reinforce the status of ISOQM as a general quality model that calls for its systematic tailoring to specific domains (as explained in Chapter 2) or at least an essential step of operationalisation to assess a 'software product' (as hinted in [Ortega et al., 2003]).

However, this attempt also reveals that the MoCQA framework is at least a good support for this operationalisation. As explained in Chapter 6, the quality assessment metamodel provides the structure needed to extract the hierarchical organisation of the quality characteristics and define properly the part of the software product that is targeted, as well as the stakeholders with an interest in this quality issue. Regarding the measurement methods, the quality assessment metamodel provides support for the specialisation of ISOQM generic metrics or the replacement of these measures by more adequate metrics. As a matter of fact, the effort made to instantiate a MoCQA version of the ISOQM makes parts of it usable (almost as a off-the-shelf components) in any other MoCQA model. The only additional process required is the more precise definition of the measurable entities and of the scope of the quality issues in order to apply a 'branch' of ISOQM to a given environment.

The customisation of ISO/IEC quality model (as well as all quality models cited in Chapter 1 that have been derived from it) is therefore a valid option for

the support of the acquisition step of the MoCQA methodology.

11.3 QualOSS documentation availability model

11.3.1 Overview

The QualOSS documentation availability model (referred to as QDAM in the remainder of this section), introduced in [Matulevicius et al., 2009], intends to provide some support for the assessment of the documentation quality, defined as *its ability to satisfy stated or implied needs of users* in ISO/IEC 14598:1999 [ISO/IEC, 1999].

Instead of focusing on a quality assessment based on the content of documentation and determining its accuracy, the quality model proposes several characteristics related to the form and determining completeness and availability of different structural parts. The QDAM thus develops a systematic assessment model that measures documentation availability according to its organisation, structural completeness and information completeness.

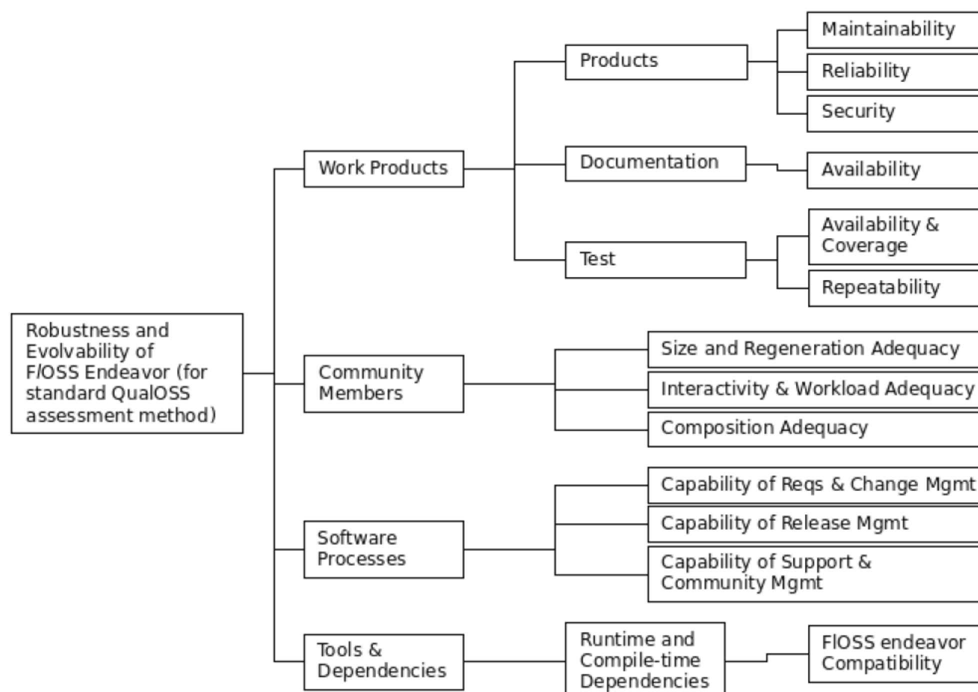

Figure 11.3: QualOSS robustness and evolvability quality model

This quality model is a part of the larger effort performed within the QualOSS project[1]. The focus of the QualOSS project has been to develop a quality model

[1] Quality of Open Source Software, project funded by European Commission under the FP6-

(Figure 11.3) designed to assess robustness and evolvability of open source software projects. This open source context explains the relevance of a documentation quality model. Indeed, in that context, stakeholders are numerous and play various roles in the project. These roles are associated to different types of knowledge and interests regarding project (e.g., users are both potential developers and/or maintainers). The documentation is therefore a crucial aspect of the development process of open source projects.

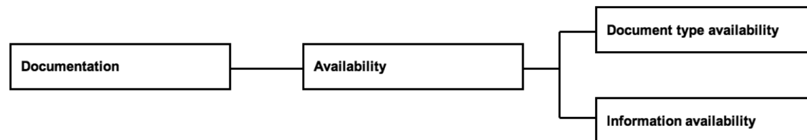

Figure 11.4: QualOSS documentation availability model

As shown in Figure 11.4, the QDAM provides the set of characteristics and subcharacteristics (respectively 1 and 2) that is expected from a quality model. Additionally, it provides the methodological guidelines to provide an indicator for these characteristics.

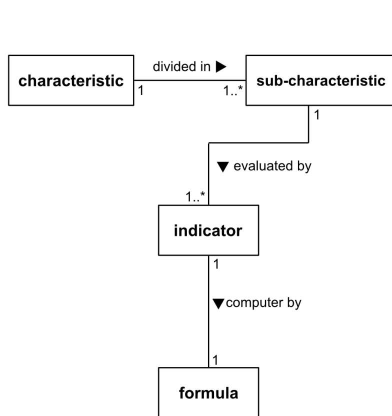

Figure 11.5: QDAM explicited metamodel

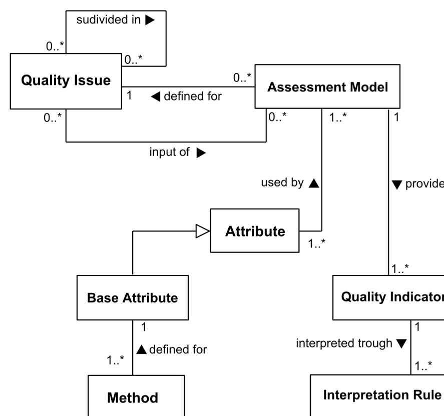

Figure 11.6: Quality assessment metamodel concepts

Figure 11.5 shows an explicited metamodel of the QDAM. The QDAM metamodel provides the same structure of characteristics and sub-characteristics as the ISOQM does. However, indicators are linked to the sub-characteristics, by way of formulas. Each indicator is associated to thresholds designed to provide an meaning to the value. Each formula relies on several base attributes to compute the related indicators. Each base attribute is evaluated by a metric. Figure 11.6 shows that concepts of the QDAM and the MoCQA quality assessment

2005-IST_5 Framework, Contract number N33547.

metamodel may be aligned. Indicators and formulas may be expressed through quality indicators and assessment models, respectively, while (sub)characteristics may be translated in quality issues. As in the case of the ISOQM metamodel, the QDAM metamodel constitutes a subset of the MoCQA quality assessment metamodel. In consequence, the quality assessment metamodel is theoretically suitable to produce an QDAM instance.

11.3.2 Metamodel Instantiation

According to the structure of the QDAM, the main quality issue of the MoCQA model is 'availability' and its scope may be defined as the 'documentation'. It is a composed quality issue, which means that the model will never produce a single quality indicator providing an assessment of the quality concept named availability. Instead, the evaluation of availability will rely on a tuple of child quality issues. The first of these quality issues is named 'information availability' and has the same scope has 'availability'. In the meantime, Figure 11.4 does not explicitly show the actual number of quality issues required. The quality characteristic 'documentation type availability' requires to scan the types of documents and will result in a different quality issue for each of them. Figure 11.7 shows that 12 types of product are considered. The MoCQA model therefore requires

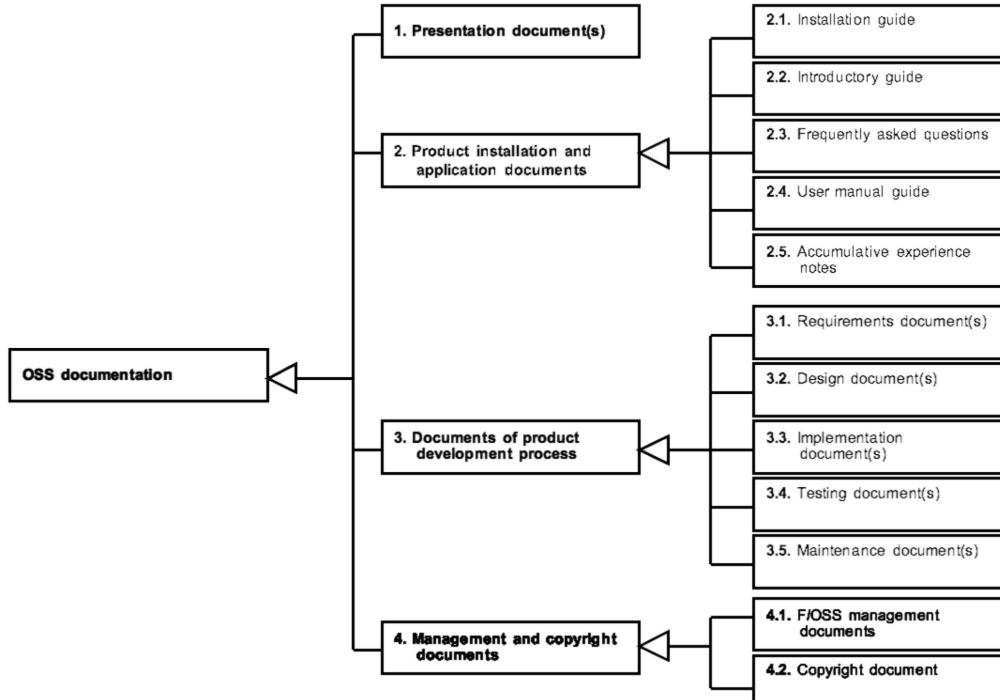

Figure 11.7: QDAM targeted artefacts

12 quality issues named 'document type availability' with a different scope for each of them (which will be the type of document listed in Figure 11.7). The resulting MoCQA model will in fact display 13 quality issues.

The QDAM defines two indicators (in the general sense),

$$DTA = \frac{DF}{DN}$$

and

$$DIA = \frac{\sum_{i=1}^{DN}(dor_i + dco_i)}{2DN}$$

that will translate into assessment models for our quality issues. The issue 'information availability' will have *DIA* as an assessment model. The 12 'documentation type availability' issues will be associated with a *DTA*. The resulting quality indicators will be associated with a ratio scale. A way to interpret the indicators is already provided by the QDAM. It states that the indicators (and thus the quality indicators of our MoCQA model) are a percentage and provides thresholds and an associated meaningful characterisation. These elements will translate directly into interpretation rules.

The project-related component is very straightforward due to the effort of clarification provided by the QDAM in terms of measurable entites. We have to define 12 resources that are all artefacts, all documentation-related and expressed in non formal written (likely English) language. The maturity level and level of abstraction are not relevant in this context.

Then, three different base attributes have to be taken into account: document organisation (*dor*), document completeness (*dco*) and an implicit base attribute that we will call 'presence' (pres). The first two will be associated with estimation methods (checklist and count of the answers) that will produce measurement values within a ratio scale and will be used in the *DIA* assessment model. The last one will be associated with a simple measurement method (i.e., the entity produces a Y or 1 if available.) that is associated with an ordinal scale (the only possible values are 'present' or 'absent'). Each *DTA* assessment model uses this attribute by counting the number of Y or 1 (*DF*, document found) with regard to the whole number of considered documents (*DN*).

11.3.3 Results

The previous section shows that it is possible to generate an instance of the quality assessment metamodel (i.e., a MoCQA model) that reflects all the aspects of the QDAM. This process of instantiation leads to several observations.

First, the QDAM as it is presents many aspects of a MoCQA model since it provides a clear definition of the targeted resources (or measurable entities) and a complete measurement method for each of them on top of a structured set

of characteristics and subcharacteristics. These (sub)characteristics may be converted almost directly in quality issues since they possess a clear scope and have just to find a suitable stakeholder in an actual context. The MoCQA counterpart of the QDAM only provides a more systematic classification of all the concepts involved as well as some further detail found in [Matulevicius et al., 2009] or simply by deduction.

Consequently, when the acquisition step relies on the customisation of the QDAM, it provides a structurally sound and coherent MoCQA model, which means that all the essential concepts, methods and relationships are provided by the QDAM and the MoCQA counterpart just provides an additional layer of information. The limitation of this method is that we cannot infer any information about the overall relevance of the QDAM in term of quality assessment. The instantiation process only concerns structural properties and does not address any semantic concern. However, the overall framework provides a valuable support for the inclusion of the QDAM in a quality assessment life-cycle.

Finally, it is interesting to notice that once this conversion step done, the examined quality model (or any part of it) can be used directly in any other MoCQA model. For instance, it would be easy to insert the *information availability* quality issue defined in the QDAM in a quality model focusing on maintenance with no risk of corrupting its relevance since all important aspects are clearly defined during the instantiation process.

11.4 Discussion

Regarding the expressiveness criterion of our validation protocol, the quality models studied were successfully translated into MoCQA models. The process did not encounter any structural problem. The main limitation of the process is the fact that without proper hypotheses on the stakeholders' requirements, quality models with a larger scope result in large MoCQA models. However, the occurrence of this issue is unlikely in a real context since stakeholders are supposed to be part of the process and help define which quality issue is required or not.

Regarding the integrability criterion of our validation protocol, the case study shows that the customisation of existing quality models is a viable method to support the acquisition step. As explained before, the MoCQA framework relies on quality models as structured catalogues of possible quality issues. The design of a MoCQA model in collaboration with the stakeholders has the stakeholders consider various options and the way they apply to their context (i.e., the need to define a scope for the quality issue). Although the process remains non trivial, the support of the quality assessment metamodel and the information needed to instantiate it allow for a better understanding of the quality model that is being used. The process is performed in a structured way, with a clear goal in mind.

This process is arguably more beneficial than a standard customisation process, since it forces the quality assurance team to adopt an operational perspective while customising the model.

Additionally, this process allows for the creation of a catalogue of reusable and independent elements that may be included in subsequent MoCQA models.

11.5 Threat to validity

A threat to validity regarding this case study is the fact that only 2 quality models were studied. Therefore, the study does not demonstrate that quality models may systematically be used as easily as an input for MoCQA models. However, the 2 quality models chosen have been selected as representative of the two main categories of hierarchical quality models. Additionally, the ISO/IEC quality model already includes many features of previous quality models. As such, the case study provides enough illustration that any quality model defined as a hierarchical structure of quality factors may potentially be translated into a MoCQA model.

A second limitation of the study is that the instantiation process has not been performed with actual stakeholders. The study therefore cannot prove or disprove that the process of acquisition would actually help the stakeholder elicit their needs more precisely in an actual context.

Chapter 12

Quality of software architecture

This chapter describes a theoretical case study that illustrates the integrability and the exploitability of the MoCQA framework at an early stage of development. The case study exemplifies the fourth acquisition method (i.e., using a complementary quality approach) described in Chapter 6 and focuses on the quality of software architecture.

12.1 Context

As explained in Chapter 6, using the MoCQA framework in conjunction with non-analytical quality assessment methods is one of the solutions to strengthen the acquisition step of its quality assessment methodology. Scenario-based quality assessment methods are especially promising approaches regarding the complementarity with the MoCQA framework in the context of software architecture.

Similarly to scenario-based approaches, the MoCQA framework allows an explicit description of the software architecture and integrates the notion of implication of the various stakeholders, making the two approaches compatible. Besides, a limitation of scenario-based methods is the fact that they are not yet integrated with metric-based approaches [Koziolek, 2011]. Such methods could therefore benefits from their integration into the framework.

Concretely, the case study described in this chapter investigates the complementarity between the MoCQA framework and the Architecture trade-off analysis method (described in Section 12.3). The case study then elaborates on a case study addressing ATAM (i.e., the BCS case study [Kazman et al., 2000]) to demonstrate the complementary between the two approaches, as well as the advantages provided by their joint use.

12.2 Objectives

This case study pursues two main goals. First, it intends to illustrate the integrability of our approach. In order to provide a good illustration of the criterion, the case study aims to show that the acquisition step of the MoCQA framework is applicable in conjunction with the selected non-quantitative quality assessment method (i.e., ATAM). It also intends to show that the integration of the two types of approaches helps limit the shortcomings of each of them. More generally, the case study aims at demonstrating how the MoCQA framework may help bridge the gap between metric-based and scenario-based approaches.

Additionally, the case study intends to illustrate the exploitability of the framework during the design phase of the development life-cycle. In order to do so, it seeks to show that the use of MoCQA helps improve software architecture quality, when used in conjunction with ATAM.

12.3 Architecture trade-off analysis method

The architecture trade-off analysis method (ATAM) is a scenario-based approach designed to reveal the level of satisfaction of an architecture towards particular quality goals such as performance or modifiability. It also helps emphasise how those quality goals interact with each other (trade-off). The final goal of an architecture evaluation using ATAM is *to understand the consequences of architectural decisions with respect to the quality attribute requirements of the system* [Kazman et al., 2000].

The ATAM is performed in 9 successive steps [Kazman et al., 2000]:

1. Present the ATAM. The method is described to the assembled stakeholders (typically customer representatives, the architect or architecture team, user representatives, maintainers, administrators, managers, testers, integrators, etc.).
2. Present business drivers. The project manager introduces what business goals are motivating the development effort and hence what will be the primary architectural drivers (e.g., high availability or time to market or high security).
3. Present architecture. The architect will describe the proposed architecture, focusing on how it addresses the business drivers.
4. Identify architectural approaches. Approaches to the architecture are identified by the architect, but are not analysed.
5. Generate quality attribute utility tree. The quality factors that comprise system "utility" (performance, availability, security, modifiability, etc.) are elicited, specified down to the level of scenarios, annotated with stimuli and responses, and prioritised.

6. Analyse architectural approaches. Based upon the high-priority factors identified in Step 5, the architectural approaches that address those factors are elicited and analysed (for example, an architectural approach aimed at meeting performance goals will be subjected to a performance analysis). During this step architectural risks, sensitivity points, and trade-off points are identified.
7. Brainstorm and prioritise scenarios. Based upon the exemplar scenarios generated in the utility tree step, a larger set of scenarios is elicited from the entire group of stakeholders. This set of scenarios is prioritised via a voting process involving the entire stakeholder group.
8. Analyse architectural approaches. This step reiterates step 6, but here the highly ranked scenarios from Step 7 are considered to be test cases for the analysis of the architectural approaches determined thus far. These test case scenarios may uncover additional architectural approaches, risks, sensitivity points, and trade-off points which are then documented.
9. Present results. Based upon the information collected in the ATAM (styles, scenarios, attribute-specific questions, the utility tree, risks, sensitivity points, tradeoffs) the ATAM team presents the findings to the assembled stakeholders and potentially writes a report detailing this information along with any proposed mitigation strategies.

12.4 Architecture analysis method with MoCQA

In order to illustrate how the MoCQA framework might take advantage of the integration between scenario-based analysis and metric-based assessment of software architecture, we applied our framework to an example of use of the ATAM.

During the case study, the specific goal has been to integrate measures that complement the scenario based ATAM evaluation. Precisely, this integration aims to:

- Provide an objective and quantitative assessment of the quality goals at an early stage of the development
- Ensure the traceability of quality aspects during the development
- Track the impact of architectural design decisions on quality
- Support the decisions regarding the design and/or the evolution of software architecture

The case study used for this illustration is the BCS case study [Kazman et al., 2000]. This evaluation addresses a system called BCS (Battlefield Control System) designed to be used by army battalions to control the movement, strategy, and operations of troops in real time on the battlefield. This system is used to

illustrate a complete example of the ATAM. Throughout the BCS study, each step of the ATAM is applied and commented.

12.4.1 Utility trees and MoCQA models

The first opportunity to integrate the ATAM into the MoCQA framework lies in the fifth step of the ATAM and, more specifically, for one of its products: the utility tree. The ATAM uses two mechanisms to elicit and prioritise scenarios: utility trees and structured brainstorming. The two mechanisms complement each other: while the brainstorming is used to consult the larger community and let its members provide input about the architecture, the utility tree provides a top-down approach designed *to translate the business drivers of a system in concrete quality attributes scenarios* [Kazman et al., 2000]. The outcomes of the two activities are compared in order to guarantee that all relevant scenarios have been considered.

Utility trees are of particular interest because they adopt a hierarchical approach, allowing the refinement of rough quality goals into more specific and concrete goals. The leaf nodes must be specific and concrete enough to allow their prioritisation relative to each other. The ATAM proposes to prioritise the concrete goals according to two aspects: On the one hand, the importance of the goal and, on the other hand, the risk posed by this goal (i.e. how the architecture team perceives the difficulty to achieve this goal). A rank (High, Medium of Low) is assigned for each of these two aspects. Figure 12.1 shows the utility tree designed for the BCS study.

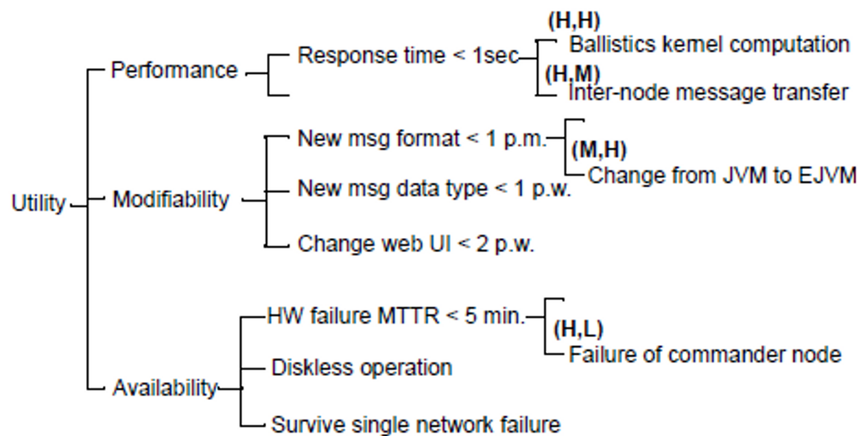

Figure 12.1: BCS utility tree

The ATAM utility tree presents the same structure as a usual hierarchical quality model. It is therefore straightforward to express this tree into a MoCQA model that preserves its semantics. Translating one branch of the utility tree

(performance branch) into a MoCQA model (see Figure 12.2), we observe that it can be completely expressed and that this can be made by means of constructs from the MoCQA metamodel quality package only. The translation relies on only a part of the constructs available in the MoCQA metamodel, and thus represents a part of the effort necessary to support a full metric-based and quality-model-based approach. The MoCQA model shows that, in order to support a metric-based approach, other inputs are needed for the assessment models.

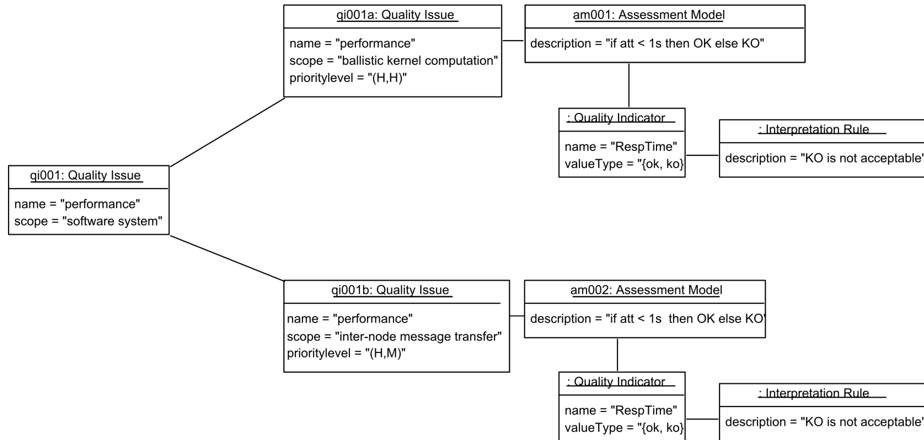

Figure 12.2: BCS utility tree expressed with MoCQA constructs

Hereafter, following MoCQA methodology, we start by proposing a first measure designed to evaluate the level of satisfaction for one of this quality goal. In order to do so, we can turn once again to the BCS case study that provides more information than those provided by the utility tree alone. The BCS case study reports that after a simple analysis (therefore not requiring any measurement), it was already clear that the performance quality was mostly influenced by its second sub-factor (inter-node message transfer). The measurable attribute regarding the performance (only referred to as `att` in the assessment model so far) should therefore be the "response time". In the followings, we therefore focus on the *quality issue* `qi001b` of the MoCQA model illustrated in Figure 12.2. This first proposal is then assessed towards the 4 goals defined in the beginning of this section before it is refined according to the assessment results.

12.4.2 First proposal of quantitative assessment

Architecture description

The BCS study mentions that the architectural documentation covers different views of the system (view of the subsystems components, sequence charts for the exchange of messages between components, etc.) as well as the highest level hardware structure which is provided in Figure 12.3. This information can easily

be expressed into a MoCQA model, each soldier or commander being an instance of the *artefact type* “node” supporting a *behaviour type* named “message transfer”.

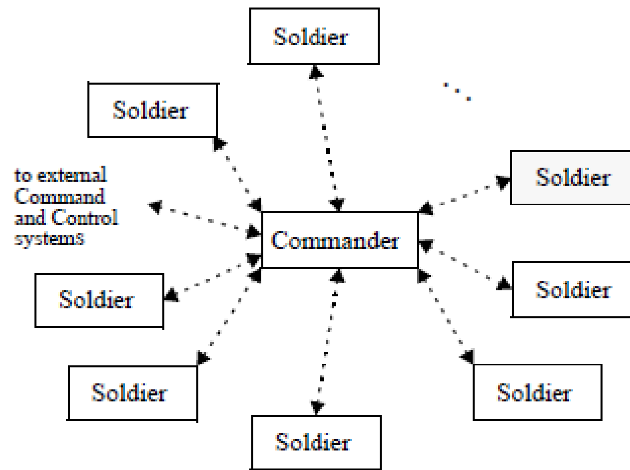

Figure 12.3: Hardware (deployment) view of the BCS

Measure definition

Now that the specifics of the scenario-based approach have been mapped into a MoCQA model, the main goal is to find relevant measures that could be used in order to show that the quality issues are satisfied. The MoCQA model, and more specifically the effort provided to model its project-related elements, constrains the choice of both an applicable and a relevant measure.

In the BCS study, considering *assessment model* `amod002` (Figure 12.2) and the fact that our current view on the architecture is a series of ‘node’ artefacts (Figure 12.3) associated with ‘message transfer’ behaviours, the most straightforward way to provide a quantitative evaluation of the satisfaction of the quality issue would be to assess directly, (i.e., by observing it), the response time for each ‘message transfer’ behaviour, as shown in Figure 12.4.

Observations

This measure is obviously an *a posteriori* control method that cannot be applied during earlier stages of the development since the software system must be fully implemented in order to be able the obtain the desired measurement values. However, it is a first step towards complementarity between scenario-based and metric-based approaches since it takes advantage of the effort made for the first one in order to provide a relevant context for the second one. Besides, it provides

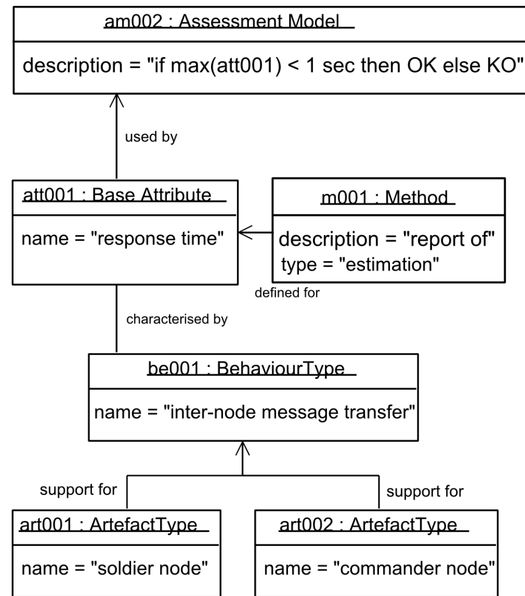

Figure 12.4: Evaluation based on behaviour types

a metric-based testing protocol destined to ensure that the quality goals (elicited through the scenario-based approach) have been achieved.

This first proposal does not appear to be fully satisfying. Our objectives are not to just acknowledge a lack of quality but to help the development team rely on the evaluation to support decisions leading to architectural quality. The measures designed to complement the utility tree should therefore avoid relying on *behaviour types* since these concepts are used to represent features of the software system at runtime.

12.4.3 Second proposal of quantitative assessment

In order to provide a more useful quantitative assessment, a first refinement aims to avoid relying on the behaviour `be001`.

Architecture description

We need to acquire more information from the BCS study and refine the way we describe the architecture in the MoCQA model. Although the BCS study does not provide such a precise information, we can extrapolate on the software architecture of each node of the BCS and consider that there is at least one class dedicated to the communication between nodes (i.e., formatting the message to send, pushing the message on the physical medium, etc.). Let's add to this

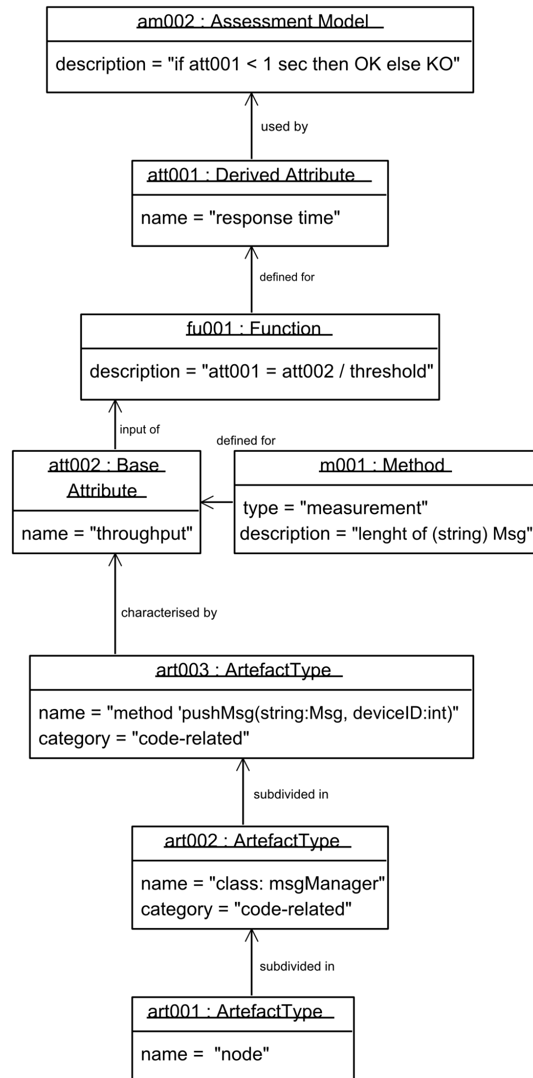

Figure 12.5: Evaluation based on code-related artefact types

description the assumption that a single method of this class has been designed to actually push the message towards the communication device.

Measure definition

As shown in Figure 12.5, this refinement of the description of the relevant part of the software project in the MoCQA model provides us with the opportunity to design a new candidate measure. This measure takes advantage of the 'msgManager' class and its 'pushMsg' method. The BCS study specifies that the sole factor influencing the response time during inter-node message transfer is the transfer

speed of the communication device that handles the transfer. Since this speed is known, it is possible to compute a threshold that represent the length in bits of the largest message that can be transferred in 1 second. Based on this threshold, we can define a measure of the response time based on the 'throughput' attribute of the 'pushMsg' method. This attribute may be evaluated thanks to the length (in bits) of the 'Msg' string that is pushed towards the communication device. By confronting this throughput with the predefined threshold, we can evaluate the response time (now a derived attribute).

Observations

This second attempt brings us closer to our objectives. The quality assessment process no longer relies on the observation of runtime behaviours. It could be used to provide support *before* implementing and testing the full software system since it helps locate the part of the architecture (i.e., the source code) where decisions have to be taken in order to avoid poor assessment results.

However this new measure is still not completely satisfying. The main flaw of the measure is that, in order to be fully efficient, it would still require testing (at least unit testing). As a matter of fact, the introduction of the threshold guarantees that we do not need to actually send messages to measure our attribute.

Still, heavy unit testing would be inevitable to get a perfect evaluation. Indeed, the throughput attribute is expected to be different for each message sent and therefore, cannot be estimated as-is on the basis of code measurement alone.

Regarding the decision support aspect, this second measure remains an improvement: it is possible for the development team to take architectural decision based on the definition of this measure. For instance, we could guarantee an acceptable throughput attribute by examining all the classes interacting with 'msgManager' and check that the messages they provide do not exceed the maximum length acceptable, according to the threshold. Another (and better) solution would be to impose constraints on the code responsible for messages formatting to ensure that any message provided to 'pushMsg' has a desirable length, even though this could require the message to be sent in multiple transmissions.

12.4.4 Third proposal of quantitative assessment

In order to provide a better support for decision purposes, we have to provide a measure that would be applicable earlier in the design process and would not require any testing in order to be computed. Once again, providing a better characterisation of our project-related constructs in the MoCQA model is the key to this refinement.

Architecture description

The BCS study mentions the existence of chart diagrams for the message transfers as well as diagrams focusing on the structure of the code (i.e., class diagrams or equivalent). These artefacts are the earliest architecture-related artefacts available. We can take advantage of the *derivation type* of our model to explicit how those artefacts relate to our quality assessment problem. We know from the previous attempt that the 'msgManager' class is the crucial part of the code regarding our quality issue. In order to justify the reason for measuring other artefacts, we have to trace their relationship with 'msgManager'. This relationship is a derivation named "implementation", meaning that in order to produce the 'msgManager' class, both the class diagram and the sequence chart have been used by the developers as a base to build upon.

Measure definition

As shown in Figure 12.6, this new refinement allows the redesign of our candidate measure. The derived attribute `att001` is still present but now relies on two base attributes. The first base attribute is the throughput of the sequence chart that we measure thanks to the length of the largest message present in the sequence chart. The second base attribute is the throughput of the 'Msg' attribute of the class diagram, that we evaluate by simple inspection of the upper boundary of the type definition. In this context, the function that links the attributes has to take into account both attributes to produce an estimation of the response time : it confronts the greater of the two values to the threshold.

Note that, in the MoCQA model, that assessment model `am002` is now a *prediction* model since it relies on internal attributes to provide a quality indicator for an external attribute. It does not actually measure the actual response time but provides a prediction of the worst response time possible, given the input values.

Observations

At this stage, the MoCQA model complements the ATAM utility tree in an efficient way. It allows the support of efficient decisions regarding the architecture (e.g., revising the sequence chart in order to limit its throughput so that the throughput of 'pushMsg' is correct and result in a good response time). Besides, all these decisions may be taken at an early stage of development. Provided that the implementation is executed correctly, these decisions will results in adequate quality aspects regarding the expectations of the stakeholders for the final product.

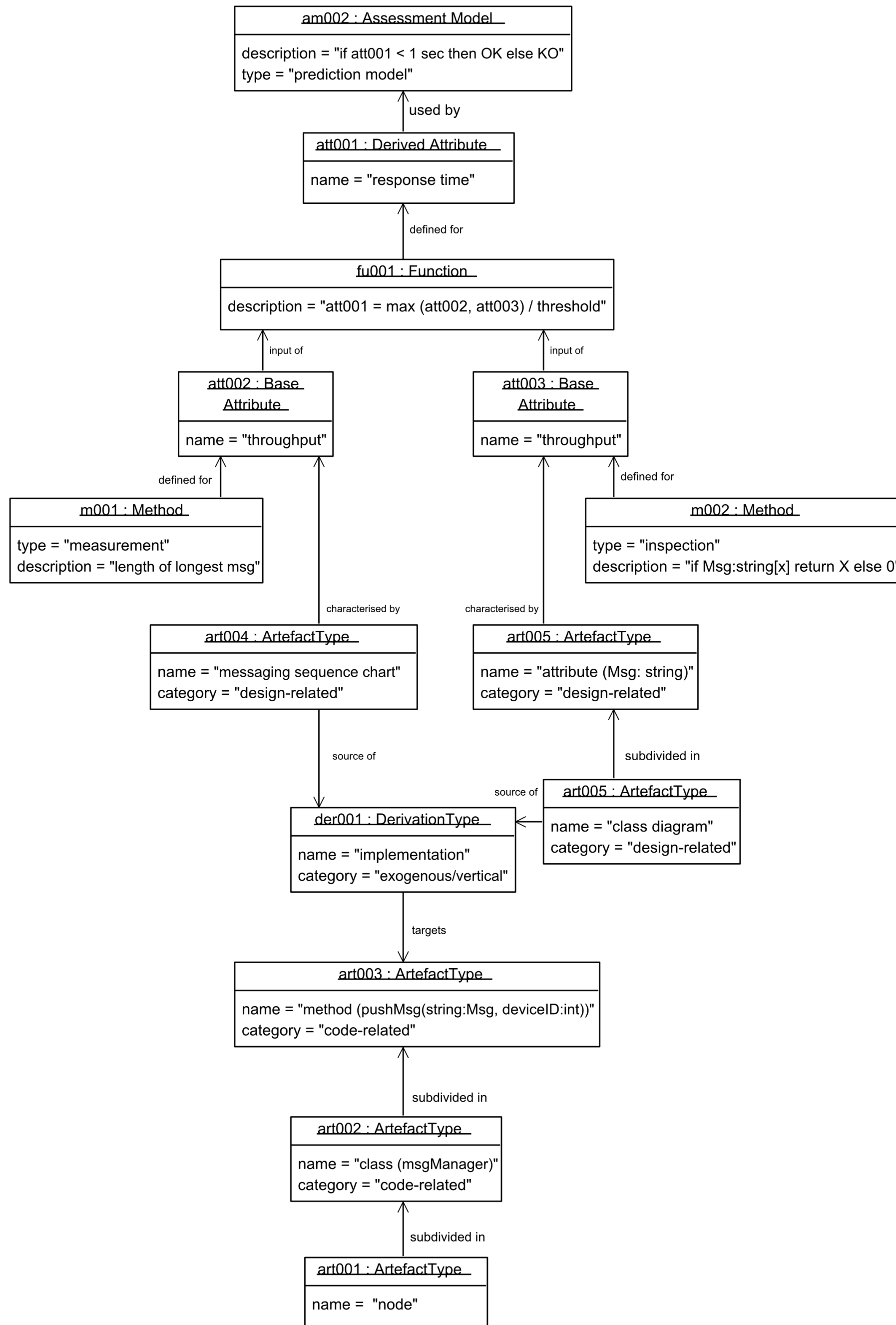

Figure 12.6: Evaluation based on design-related artefact types

12.5 Results

As illustrated in Section 12.4, it is possible to support the ATAM with metric-based methods thanks to the MoCQA framework. This section illustrates the benefits revealed by the case study regarding this integration in the context of software architecture quality.

12.5.1 Support for utility tree processing

As explained in Chapter 9, the XOCQAM language provides a better support for MoCQA models to be easily queried and analysed. Regarding utility trees, once integrated in a XOCQAM document, the possibility to query the model becomes a key advantage for the subsequent analysis. For instance, the ability to easily filter quality issues and their dependent elements (attributes, measurable entities, etc.) by priority level allows us to easily guide the measurement process as it occurs. Similarly, the ability to easily (and automatically) detect potential conflict between quality issues (e.g., two quality issues depending on the same attribute but associated to two quality indicators that require opposite values for this attribute) represent an efficient mechanism to anticipate poor results of any trade-off.

12.5.2 Support for quality traceability

Section 12.4 also shows that the approach provides a good insight regarding traceability of quality aspects along the development. In our case study, the use of measures #2 (Section 12.4.3) and #3 (Section 12.4.4) provides a good example of such insights. For instance, during an hypothetical maintenance of the system illustrated in Section 12.4, if the performance of the system were to be perceived as unsatisfactory, measures #2 and #3 would help identify which part of the architecture has to be refactored.

Indeed, if measure #2 and #3 display poor results, one can deduce that either the class diagram or the sequence chart should be revised in addition to the 'msgManager' class. On the other hand, good results with measure #2 coupled with poor results with measure #3 would indicate that the refactoring process should focus solely on the class and point out that the implementation is the step where quality aspects were lost. If both measures display satisfying results, the problem may be linked to the physical medium itself. This allows a better understanding of how quality aspects evolves through the various level of abstractions constituting the architecture and through time. Therefore this approach provides a complementary viewpoint to the rationale recording principle [Budgen, 2003] and could therefore be used in conjunction with existing frameworks focusing on this aspect (as in [Gilson and Englebert, 2011]) to provide a total traceability of architectural rationale, design decisions and quality aspects.

12.5.3 Support for architecture refactoring decisions

The integration of the metric-based and scenario-based approaches present a good potential regarding the motivation of subsequent decisions regarding model refactoring. The scenario-based approach provides a precise definition of the quality goals while the metric-based approach provides information on how those goals will be monitored. The MoCQA model provides an integrated mechanism to keep track of all this information. In our case study, the MoCQA model shows useful in order to make sound decisions (e.g., measure #3 hints at the necessity to address the size of messages in the sequence chart and the class diagram in order to constraint the implemented code). As explained in Chapter 8, the measures used in the context of a MoCQA model do not always need to provide actual measurement results in order to prove useful: the awareness of their existence and the way they are computed may suffice to lead to good architectural decisions.

12.5.4 Support for architecture design decisions

As explained in Chapter 7, the third step of the MoCQA methodology (tailoring of the measurement plan) is necessary in order to actually use the MoCQA model. This step requires the identification of measured entities and the definition of concrete measurement *procedures* to support the conceptual measurement *methods*. This process can be automated to a certain level: if the name of the artefact type is simple, tagging all occurrences of this type among all the resources of the project as measurable entities can be done automatically with an adequate tool-support (e.g., Java class, Java class name X). In the case of more complicated properties (e.g., Java classes supporting behaviour X), a manual inspection of the code remains necessary in order to tag the adequate resources. However, the pay-off for this task is worth it. In conjunction with the MoCQA model, team members have the opportunity to be constantly aware of the quality issues as they refactor or implement a tagged resource. They can therefore estimate the potential impact of their decisions and actions on the overall quality of the software system. In the case of software architecture quality, this property proves even more useful. In our case study, the ability to know the maximum length of the transferred messages as the coding occurs is a crucial element contributing to the overall quality of the software project.

12.5.5 Flexibility of the approach

The iterative approach of the MoCQA framework provides a way to be flexible regarding quality assessment. For instance, measure #3 requires an effort to plan the properties of the code during the design (i.e., decide that the code *will* contain a class named ‘msgManager’ and a method named ‘pushMsg’). If a different architectural choice appears during the implementation phase, the

MoCQA model will be revised in order to adapt to the architecture change during the next quality cycle, ensuring that the traceability of the quality aspects is still possible.

12.6 Discussion

In this chapter, we investigated the potential of the Model-Centric Quality Assessment framework to support an efficient quality assessment of software architecture through the integration of a scenario-based evaluation method and metric-based assessment.

Our case study using the architecture trade-off analysis method (ATAM) shows a satisfying potential to achieve this goal. It therefore demonstrates the *integrability* of the MoCQA framework in an context where the ATAM is used.

Regarding the *exploitability* of the framework, and especially at early stages of the development, the case study provides several positive elements. Provided that the measures are defined on the basis of a relevant structured description of the architecture and that an adequate tool-support is provided, MoCQA models offer a good support for:

- Providing an objective and quantitative assessment of the quality goals at an early stage of the development.
- Ensuring the traceability of quality aspects during the development.
- Tracking the impact of architectural design decisions on quality.
- Supporting the decisions regarding the design and/or evolution of the software architecture.

Therefore, the results tends to show that the MoCQA framework is exploitable in this context.

12.7 Threat to validity

Although the results should be considered relevant regarding the two criteria it aims to assess (i.e., integrability and exploitability), several limitations of the case study itself have to be taken into account in their interpretation.

First, our case study limits its scope to the ATAM approach. Although they share the same core principles, other scenario-based methods to quality assessment (such as the ALMA method [Bengtsson et al., 2002]) should be investigated in order to generalise the results of the case study.

Another issue is the theoretical nature of the case study itself. First and foremost, the use of the BCS case study as a basis for this case study implies a lack of information on some important elements of the actual architecture. Although the assumptions made regarding the structure of the code sounds reasonable, their

speculative nature remains. Going past the simple positive reinforcement and actually proving the exploitability of the framework at early stage still necessitates practical studies in actual contexts

The final issue is the scalability of the approach. On more complex cases, the approach could be more demanding regarding the effort to produce a MoCQA model with the right level of detail. This aspect will be improved through better tool support. As hinted by the case study, the more the tool support will be refined, the better we will be able to take advantage of the MoCQA framework during software development.

Chapter 13

Empirical studies

This chapter compiles three empirical studies performed in an academic context during the course of this research. These case studies illustrate the adaptability, the exploitability and the expressiveness of the MoCQA framework. Therefore, they also exemplify concepts described in Chapter 5, 7 and 8.

13.1 Preliminary study

13.1.1 Context and objectives

This small exploratory study (reported in [Vanderose et al., 2010]) was designed as a series of usability and acceptance tests for the quality assessment metamodel. In the context of a software quality assignment, each student was asked to provide a series of MoCQA models designed to solve theoretical assessment-related problems. The students were therefore inexperienced developers with little to no knowledge about quality assurance. No actual assessment was performed on the basis of these MoCQA models. However, each test (i.e., each quality-related problem proposed to the students) was designed to represent a different category of problems, calling for the use of specific constructs of the quality assessment model. The goal of each test case was to assess the ability of students to provide a syntactically and semantically correct MoCQA model that fitted the specific context of the hypothetical quality-related problems. Regarding our validation process, the criteria illustrated by this study is thus the expressiveness of the framework.

13.1.2 Description

Before the tests were performed, the students were given the quality assessment metamodel and basic explanations on the semantics of the constructs. For each test, 8 groups of 2 students were confronted to a set of ten non trivial assessment-related problems for a given medium-sized software project (e.g., quality of the implementation of the design, level of completion of the project, robustness of non functional aspects of the project, etc.). The aim was to design a MoCQA model to structure and streamline each of those assessment-related problems. Using the quality assessment metamodel, the groups were thus asked to propose a hierarchy of quality issues as well as measurement or estimation methods to monitor them and to characterise the measurable entities. The relevance of their MoCQA model was assessed through expert advice (i.e., the advice of the teacher and teaching assistant).

13.1.3 Results

At the end of the assignment, all the designed MoCQA models were assessed by the teacher and teaching assistants. For each MoCQA model, the evaluators checked that:

1. The semantics of each construct of the metamodel was respected in the instantiated model;
2. The MoCQA models were structurally valid (i.e., that no infringement regarding the associations between constructs was detected);
3. No violation of the content integrity (Chapter 8) was detected.

The *relevance* of the quality assessment process that had been modelled was not taken into account in the validation process.

Regarding these aspects, the designed MoCQA models of the 8 groups were considered correct by the evaluators. Indeed, for each test case, students were able to provide a syntactically and semantically valid set of MoCQA models.

Due to the small available data set, no quantitative data was collected during this case study since the statistical relevance would have been insufficient. However, the students were informally interviewed at the end of the study. The aim of the interview was to report the problems they encountered during the design step. 6 groups out of 8 reported that the main hindrance was to select adequate evaluation methods to integrate into the MoCQA model.

13.1.4 Discussion and threat to validity

This study should be regarded as a preliminary testing of the quality assessment metamodel. The fact that the students were inexperienced regarding software quality but were able to provide coherent (although not always relevant)

MoCQA models tends to show the expressiveness of the framework. Indeed, the set of constructs provided by the quality assessment metamodel were sufficient and semantically well-defined enough to allow the subjects to start applying the approach.

Although a controlled experiment is still required in order to demonstrate such statements, the results of the interviews performed during this case study tend to show that the framework help newcomers understand quality and software measurement concepts more easily due to the fact that they experiment visually with these concepts while achieving their MoCQA model.

Besides, the fact that a majority of students experienced difficulties regarding the selection of assessment and evaluation methods tends to show the relevance of tools such as QuaTALOG.

A threat to validity of this study is the small number of subjects that participated in the study as well as the lack of formalisation of the data collection. However, as an early testing protocol, this study should be considered positive regarding the expressiveness of the framework.

13.2 Preliminary study: Quality of OSS

13.2.1 Context and objectives

Another exploratory study was conducted with students in the context of the same software quality course. For this study, 9 groups of (2-3) students were asked to assess and compare 2 open-source and 1 commercial productivity suites using the MoCQA framework. As for the previous study, the students were inexperienced developers with little to no knowledge about quality assurance. A complete quality assessment cycle was performed by each group. The goal of the study was to check that students would be able to perform the quality assessment cycle, from the elicitation of the quality requirements (on two hypothetical cases) and towards a sound comparison leading to a decision regarding the proposed case. Another objective was to determine if the 9 groups would come to similar (or at least non contradictory) conclusions. Regarding our validation process, the criteria illustrated by this study are the expressiveness, the adaptability and the exploitability.

13.2.2 Description

Before the study was performed, the students were given access to a MoCQA deployment guide. Two hypothetical organisational contexts (complete with hypotheses on the budget constraints, number of employees, etc.) were defined. For each context, groups were assigned the task to provide MoCQA models to assess the fittest productivity suite among three options (i.e., MS Word, Open

Office, Libre Office), according to the hypothetical context-related requirements. The acquisition step was performed with one teaching assistant acting as the representative of the organisation described. Once the quality assessment cycles performed, groups were invited to present and discuss their results in front of each other. Additionally, the relevance of their models and interpretations were validated through expert advice (i.e., advice from the teachers).

13.2.3 Results

During the course of the study, each group was able to deploy the framework and produce MoCQA models (one for each productivity suite) that were considered syntactically and semantically valid, according to the same validation process as the previous case study. However, the relevance was taken into account this time. The models were judged adapted to the quality-requirements by the 4 experts.

The hypothetical context in which the assessment took place was designed by the teachers so that only one of the assessed productivity suites should be regarded as the fittest solution (i.e., Open Office). During the course of the study, the teaching assistant acting as stakeholder was in charge of providing this information by answering the questions of the students. At the end of the assessment process, 7 out of 9 groups provided the decision expected by the evaluators on the basis of their MoCQA model.

No additional quantitative data was collected during this case study.

13.2.4 Discussion and threat to validity

This study should also be regarded as an early testing of the MoCQA framework, focused on the exploitability, expressiveness and adaptability.

Similarly to the previous study, the fact that the groups were able to provide syntactically and semantically valid MoCQA models tends to show the expressiveness of the framework. The fact that the MoCQA models were judged relevant for the context tends to show its adaptability.

The fact that 7 groups provided the expected assessment reinforces the confidence regarding the exploitability. Although 2 groups drew controverted conclusions, the reason was identified as the inclusion of irrelevant external factors (such as personal opinions of the students, hypotheses regarding the future of one of the productivity suites, etc.) and not related to the assessment of the software products themselves.

This study shares the same threats to validity as the previous one (i.e., a small number of subjects and a lack of formalisation of the data collection). However, as an early testing protocol, this study should also be considered positive regarding the expressiveness, adaptability and exploitability of the framework.

13.3 Support for software maintenance and evolution

This section describes an empirical study performed on student projects. The study investigates the potential of the MoCQA framework to be used as a support of the maintenance and evolution of software. As such, it provides an illustration of the exploitability, the adaptability and the expressiveness of the framework. The empirical study follows the guidelines on how to conduct an empirical study proposed in [Wohlin et al., 2000] and also illustrates how the steps defined by these guidelines match the steps of the MoCQA methodology. The remainder of this section is an extended version of [Vanderose et al., 2012].

13.3.1 Planning of the study

Concretely, the study we performed was designed to show that the way measured entities are defined (and therefore identified during the measurement process) influences the accuracy of the quality assessment performed with these measures. Secondly, the study intended to show that ill-defined entities prevent measures to achieve their potential as support for the maintenance of software projects.

The planning of the study has been defined as follows:

1. Selection of candidate software projects to evaluate
2. Design of a MoCQA model/of customised measures
3. Identification and classification of the measurable entities
4. Application of the measurement procedures
5. Analysis and evaluation of the results

The above process was repeated twice. First, we defined two customised measures (that we translated into a simple MoCQA model for comparability) and we assessed their potential to guide the maintenance process. Then, we repeated the study with an improved (refined) version of the MoCQA model in which the connections between software artefacts were explicitly taken into account. We then observed how the refinements impacted the usefulness of the measures regarding the maintenance process.

The quality characteristic that we focused on is *completeness*. Although the notion of completeness appears intuitively relevant as far as quality is concerned, this characteristic does not appear explicitly in ISO/IEC 9126 quality model [ISO/IEC, 2001a]. As shown in Chapter 1, the *completeness* characteristic originates from McCall's model [Mccall et al., 1977].

The choice to focus on completeness instead of one of the characteristics originating from the ISO/IEC 9126 standard is due to the context of the study. The evaluation of student projects implicitly relies on the correction and completeness attributes. McCall's model is therefore adapted to the quality assessment in an academic context.

This step corresponds to the quality requirements elicitation activities of the *acquisition step* of the MoCQA methodology.

13.3.2 Design of the study

Selection of projects

The selection of projects for the experimental study was based on 3 criteria:

1. each project should have the same set of functionalities
2. required documentation should be available
3. an existing quantitative evaluation should be available

These criteria lead to the selection of 6 medium-sized student projects. These project had been developed in the context of a software engineering assignment, by teams of 4 to 5 students having little practical experience. However, the students had already acquired the skills needed at the theoretical level (i.e., database engineering, UML modelling, Java programming, etc.).

The course itself attempts to simulate a plausible real-world complete software life-cycle, from requirements engineering to the maintenance phase. In this context, the teacher and 3 teaching assistants act as manager and senior consultants with 2 other teaching assistants acting as clients. The latter are independent from the software engineering course (i.e., coming from a different research centre). During the assignment, the students are given three months to implement a complete stock exchange application. In this context, the resulting projects are expected to present major flaws.

The availability of an existing quantitative evaluation is therefore guaranteed since the 6 projects are evaluated in the context of the usual course evaluation. However, this evaluation does not rely on software measurement. Instead, the evaluation is conducted on the basis of expert inspection (the teacher and 3 teaching assistants) and client satisfaction (2 teaching assistants). Additionally, the projects are evaluated through a testing phase based on a specific test plan developed by the teaching team and unknown to the students. The outcome of this testing phase is also used as a basis for a subsequent maintenance phase of the projects. Guidelines to conduct the maintenance phase are provided by the teaching team after analysis of the results.

In this study, we used the evaluation and guidelines issued from the course evaluation as a control mechanism to ensure that the results provided by the measurement methods were effectively coherent with the evaluation of the projects.

During this step, we gathered knowledge on the environmental factors of the investigated context, therefore completing the *acquisition step* of the MoCQA methodology.

Design of MoCQA models

In order to create a hierarchy of quality goals of our completeness MoCQA models, we translated the relevant factors of McCall's quality model to provide a hierarchy of quality issues. However in order to fit our specific quality-related needs, we had to refine the completeness quality issue.

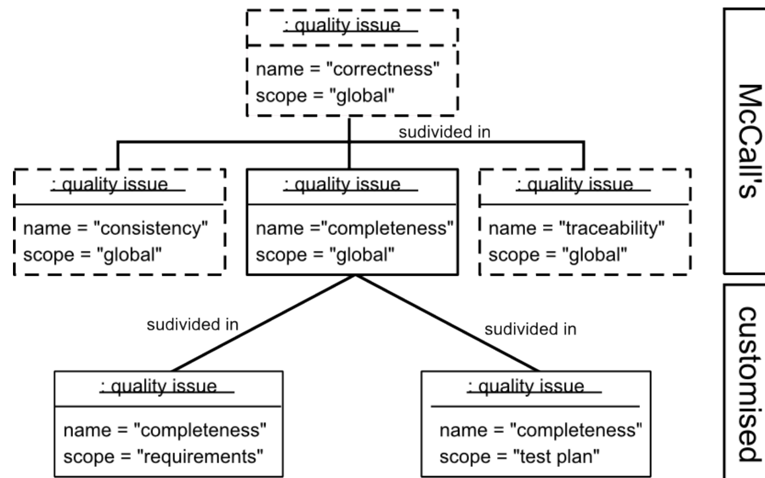

Figure 13.1: Completeness quality factor decomposition

As Figure 13.1 shows, the global completeness quality issue has been refined in two sub-issues: completeness of the requirements and completeness of the test plan. In the context of an instantiation of the full McCall's quality model, the *completeness* quality issue would therefore be a sub-issue of the *correctness* quality issues. We disregarded other potentially important completeness factors, such as code completeness and design completeness. The main reason for ignoring these factors was to keep the exploratory study manageable.

Note that McCall's completeness is not applicable to the context in the way it was originally defined. First, our main goal is to provide objective measures for the assessment of the projects whereas McCall's defined metrics are often dismissed as being too subjective [Ortega et al., 2003].

Secondly, McCall's model has been designed from a strict software product point of view, that is, focused exclusively on the source code. In contrast, the student projects in our study are complex model-driven projects that require the inspection of other aspects (i.e., requirements, test plan, etc.)

Therefore, the MoCQA framework was used to redefine a customised model using McCall's hierarchy of quality factors/criteria with more objective measures specifically defined for the context.

During our study we successively assessed two different completeness MoCQA models and their associated measures: we started with a simple, coarse-grained

version, followed by a refined, more complete, and more fine-grained version that designed to provide more exploitable results.

First version of completeness MoCQA model

For the first MoCQA model of our study, we actually designed two customised measures *before* we translated them as MoCQA constructs. As such, we did not follow the standard procedure described in Chapter 5. We therefore did not benefit from the description of the project-related component of the MoCQA model during the design of the measures. The two measures were designed logically but do not rely on the specific constructs of the quality assessment metamodel. Once the measures were defined, we translated them into MoCQA constructs and documented the project-related component afterwards. The resulting MoCQA model is described below.

The **project-related** definitions of our first MocQA model are shown in Figure 13.2. They focus on 5 types of artefacts. Two main measurable entities are defined: "requirements" and "test plan". Those measurable entities are defined in the MoCQA model as requirement-related and test-related artefact types, respectively. Those two artefact types are large and thus possess a coarse level of granularity. The three other defined measurable entities display a higher level of detail: "use cases", "use case scenarios" and "test cases". These artefact types are measured as collection of entities, as explained in Chapter 5.

The **measurement-related** definitions translate the customised measures we designed prior to the MoCQA model. These measures had to remain simple in order to verify that an increased level of accuracy (in the refined model) would be due to more accurate descriptions of the entities. We chose to define simple ratios inspired by SQuaRE's internal quality measurement methods [ISO/IEC, 2011]. All the measurement methods defined are thus mere counting methods (for the base attributes) associated with one measurement function that provides a ratio of these two values. In Figure 13.2 and 13.3, the measurement methods are hidden in order to keep the figures more legible.

As explained in Chapter 5, the definition of the measures in the MoCQA model assumes that the measurement methods will be applied to each existing measurable entity type conforming to the definition within the software project environment. The measures defined for the "requirements" and "test case" are applied to only one instance of each artefact since those types of artefacts have unique instances.

As shown in Figure 13.2, the first measure we defined addresses the completeness of "requirements". This measure, *comp-req*-1, is translated as a derived attribute evaluated through the number of "use cases" (UC) and the number of

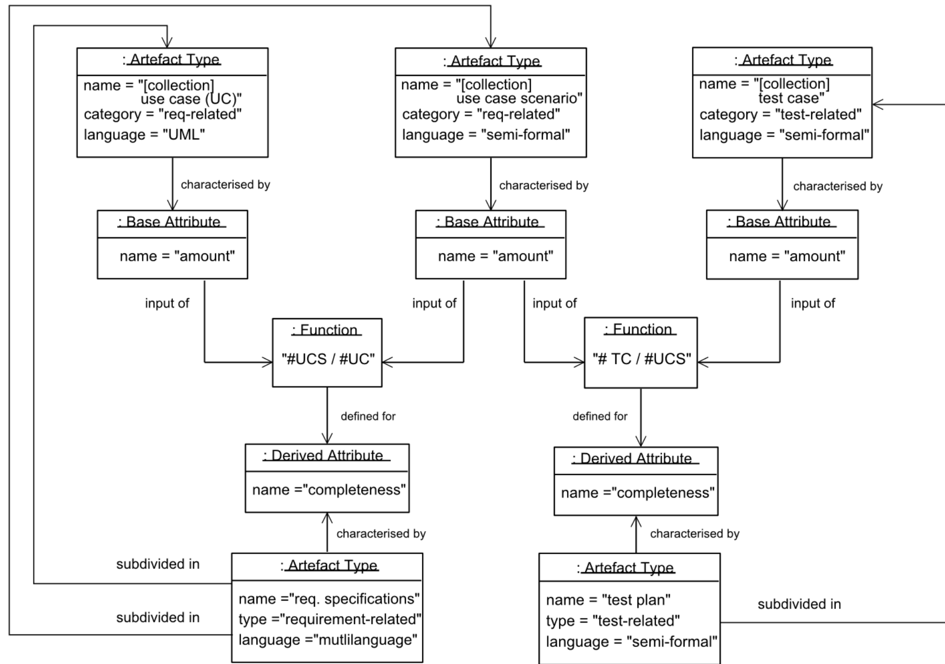

Figure 13.2: Basic MoCQA model (measurement and project components)

"use case scenarios" (UCS):

$$comp\text{-}req\text{-}1 = \frac{\#UCS}{\#UC}$$

This measure has a very simple measurement method and a very simple definition of entities. No link between the use cases and the scenarios is defined.

We also defined a measure that focuses on the completeness of the "test plan". This measure, *comp-test*-1, is a derived measure based on the number of "use case scenarios" and the number of "test cases". The rationale behind this evaluation is to check that enough test cases have been defined to cover all functionalities. The function comp-test-1 is defined as:

$$comp\text{-}test\text{-}1 = \frac{\#TC}{\#UCS}$$

This measure also has a very simple measurement method and a very simple definition of entities. As with *comp-req*-1, no evidence of any link between a given use case scenario and a given test case is provided in the definition of the entities.

Second version of completeness MoCQA model

Figure 13.3 illustrates a refined version of the completeness MoCQA model. This model was designed according to the standard procedure (i.e., the design of the

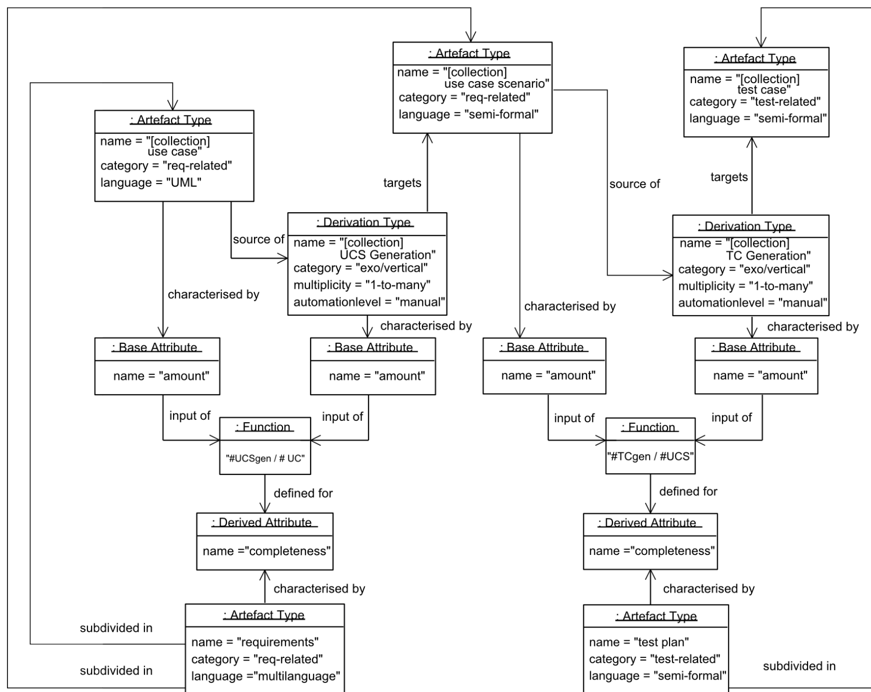

Figure 13.3: Refined MoCQA model (measurement and project components)

project-related component prior to the measurement-related component). Compared to the previous version, it adds two new measurable entities that are defined as derivation types.

The first derivation type is named “Use Case Scenarios (UCS) generation”. It has a *1-to-many* multiplicity since to each use case correspond many scenarios with alternate cases. The derivation type is exogenous and vertical, since the goal of the derivation is to provide a lower level of abstraction for each use case (according to the taxonomy proposed in [Mens and Gorp, 2006]). The level of automation is identified as *manual* since no automation of any kind has been used in the projects.

The second derivation type is named “Test Case (TC) generation”. It captures the fact that to each use case scenario corresponds at least one (but usually more than one) test case to cover the functionality. The characterisation of the derivation type is similar to the previous one, only with different sources and targets.

Based on this improved completeness MoCQA model, we provided an improved version of the completeness measures *comp-req*-1 and *comp-test*-1. This time, the definition of the measures benefit from the modelling effort consented regarding the measurable entities. In contrast with the previous ones, these two measures have been refined using the specific constructs of the MoCQA model

(derivation types) that helped define more precisely what we really intend to measure.

Measure *comp-req*-2 is an improvement over measure *comp-req*-1 designed to assess the completeness of "requirements". It is defined as a derived attribute evaluated through the number of "use case scenario generations" (UCSgen) and the number of "use cases" (UC):

$$comp\text{-}req\text{-}2 = \frac{\#UCSgen}{\#UC}$$

This measure still has a simple measurement method but the entities measured have been described in more detail than for *comp-req*-1. Nevertheless, it remains a naive vision of the problem compared to more sophisticated completeness models such as [Firesmith, 2005].

Measure *comp-test*-2 is an improvement over measure *comp-test*-1 to assess the completeness of the "test plan". It is defined as a derived attribute based on the number of "test case generations" (TCgen):

$$comp\text{-}test\text{-}2 = \frac{\#TCgen}{\#UCS}$$

Measurement plan

Due to fact that the defined measures were relatively simple and to the fact that the person in charge of the measurement process also participated to their design, no particular efforts were made regarding the formalisation of the measurement plan. The only caveat relates to the correct identification of which measurable entities are to be considered due to the introduction of a derivation type. The measurement procedure has to take into account that counting the defined derivation (for the requirement completeness measures) translates as the following algorithm:

```
UCSGen:= 0
UCSet:= Set of all existing UC
UCScenSet := Set of all existing UCS
WHILE UCSet not empty DO
    CurrentUC := one element of UCSet
    IF 1 elt linked to CurrentUC is found in UCSScenet THEN
        UCSGen:= UCSGen + 1
    Remove CurrentUC from UCSet
```

If more formalisation had been required, one of the derivation type removal techniques described in Chapter 7 could have been applied.

Additionally, all measurement procedures were applied manually and verified several times. This was made possible by the relatively small scope of the measurement process. In the case of bigger projects, the measurement procedure could have relied on a specific device (such as a traceability matrix) to perform the measurement more easily.

Regarding the quality indicators, no assessment model was defined during this study since the goal was to investigate the measures and not to provide an indicator *per se*. However, a global completeness indicator was computed afterwards as part of the analysis of the measurement data, for comparison purpose (see Section 13.3.3).

13.3.3 Experimental study

Analysis of results for the first version of the completeness model

The measurement values collected during our study are presented in Table 13.1. For each of the 6 student projects, the measurement values of *comp-req*-1 and *comp-test*-1 are provided.

	comp-req-1	comp-test-1
Project 1	1.02	2.41
Project 2	1.00	1.83
Project 3	0.77	3.24
Project 4	1.97	1.46
Project 5	1.00	1.63
Project 6	0.94	1.00
Average	1.12	1.93
Standard deviation	0.43	0.79
Median	1	1.73
Outliers	2	0

Table 13.1: Measurement values from the first version of the MoCQA model

According to the design of the functions defined for the derived attribute (a basic ratio between base attributes) the measurement values should be interpreted as percentages, meaning that a value of 0.75 should be understood as “75% complete”. However, the results from Table 13.1 can clearly not be interpreted this way since *comp-req-1* and *comp-test-1* display overoptimistic completeness assessment exceeding 100% completeness. Without a better interpretation of the measurement values, those values would be misleading or useless.

A deeper analysis of the measurement definitions *comp-req*-1 and *comp-test*-1 provides an explanation. The measures *comp-req-1* and *comp-test-1* attempt to produce a ratio of two unrelated types of entities. The amount of use case scenarios *is* supposed to be more important than the amount of use cases *by design*, since

any use case is refined into several scenarios (the same applies to the relationship between scenarios and test cases). The ratio between the two amounts will almost always produce values exceeding 1, except for very incomplete projects (such as projects 3 and 6). Therefore, measures *comp-req-1* and *comp-test-1* should not be interpreted as percentage but as the rather imprecise "anything under 1 is not good" rule.

Analysis of results for the second version of the completeness model

The measurement values collected using the second version of the MoCQA model are given in Table 13.2. For each of the 6 student projects, the measurement values of *comp-req*-2 (improved version of *comp-req*-1) and *comp-test*-2 (which improves upon *comp-test*-1) are provided. These measures provide more realistic values.

	comp-req-2	comp-test-2
Project 1	0.72	0.58
Project 2	0.76	0.54
Project 3	0.77	0.84
Project 4	1.00	0.62
Project 5	0.71	0.86
Project 6	0.74	0.62
Average	0.78	0.68
Standard deviation	0.11	0.14
Median	0.75	0.62
Inter Quartile Range	0.04	0.19
Outliers	1	0

Table 13.2: Measurement values from the improved version of the MoCQA model

Since all the measures were designed with the same basic formula (i.e., a ratio between two amounts of entities), the difference between the two couples of measures has to be related to the choice of the entities themselves and, therefore, to the accuracy of their definition (i.e., project-related modelling).

In this case, the definition of the "UCS generation" derivation type (and the "TC generation" derivation type) makes it possible to provide more precise measurement definitions. By focusing on the generation of scenarios (or test cases) based on each use case, measures *comp-req-2* (or *comp-test-2*) count the amount of *use cases that are actually covered in the specifications (or tests).* The ratio between this amount and the amount of use cases makes more sense and may be used to provide a correct completeness indicator, that may be interpreted as a percentage and thus provides a more accurate quality assessment.

Comparing both versions

Figure 13.4 shows the boxplots for each measure, allowing the comparison of the distributions of values obtained for each measure. A first observation we can make is that there is much less variation for the measures based on the second version of the MoCQA model.

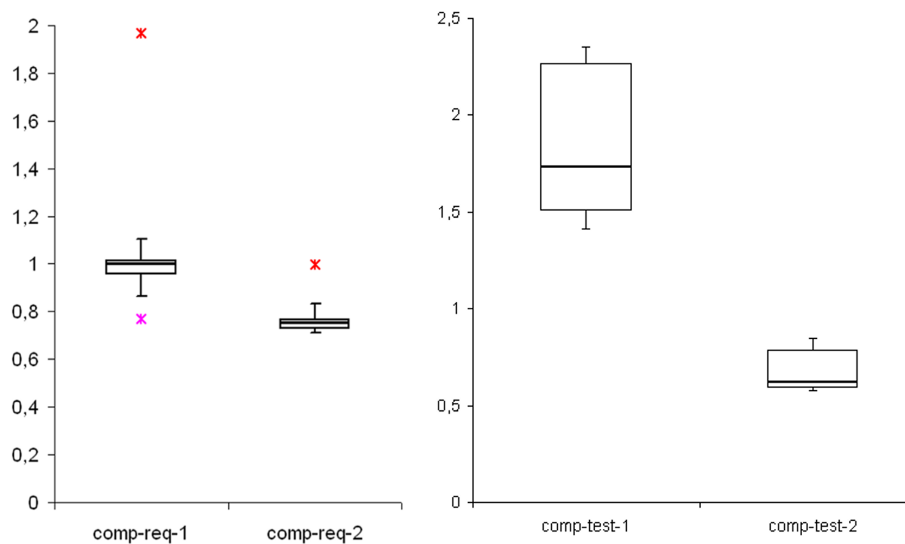

Figure 13.4: Boxplots for each of the 4 measures

Table 13.3 provides the values for the global completeness indicators. The global completeness indicators have to take into account the evaluation (estimation or measurement) of requirements and test plan completeness (Figure 13.1). The value *comp-v3* provides a numerical representation of the teachers' evaluation as a percentage. This value has been computed on the basis of the scores given to the groups at the end of the assignment. However, the score was modified in order to avoid the influence of all the factors unrelated to requirements or test plan (i.e, the scores of unrelated aspects have been subtracted from the total score before converting it into a percentage). Therefore, this can be assimilated to a global completeness indicator.

In order to construct a global indicator on the basis of the measures defined in Figure 13.2 and 13.3, we have to define the relative importance of requirements completeness and test plan completeness. Moreover, this definition of weights must be compatible with the definition of *comp-v3* in order to allow the comparison between the indicators. In the computation of *comp-v3*, the emphasis on requirements is equal to the emphasis on the test plan. Therefore, the value *comp-v1* is the average of *comp-req-1* and *comp-test-1* and the value *comp-v2* is the average of *comp-req-2* and *comp-test-2*. As shown in Figure 13.5, the values

	comp-v1	comp-v2	comp-v3
Project 1	1.72	0.65	0,60
Project 2	1.42	0.65	0.60
Project 3	2.01	0.81	0.60
Project 4	1.72	0.81	0.67
Project 5	1.32	0.79	0.67
Project 6	0.97	0.68	0.75

Table 13.3: Global completeness indicators

of *comp-v1* are totally inconsistent with those of *comp-v3*, which is not surprising since the values are not in the same range due to the poor definition of the entities (they are not percentages as explained in Section. 13.3.2). Regarding *comp-v2*, the consistence with *comp-v3* is not perfect either. Although the values are closer (around 0.70), the two indicators do not provide a similar ordering for the student groups. This inconsistency can be explained by the fact that the subjective evaluation remains more prone to detect number of flaws in the design of the scenarios (or test cases), explaining why the scores are generally lower than the indicators. Therefore, the careful examination of the project remains a better way to provide accurate quality assessment but the customised measures provide a good approximation.

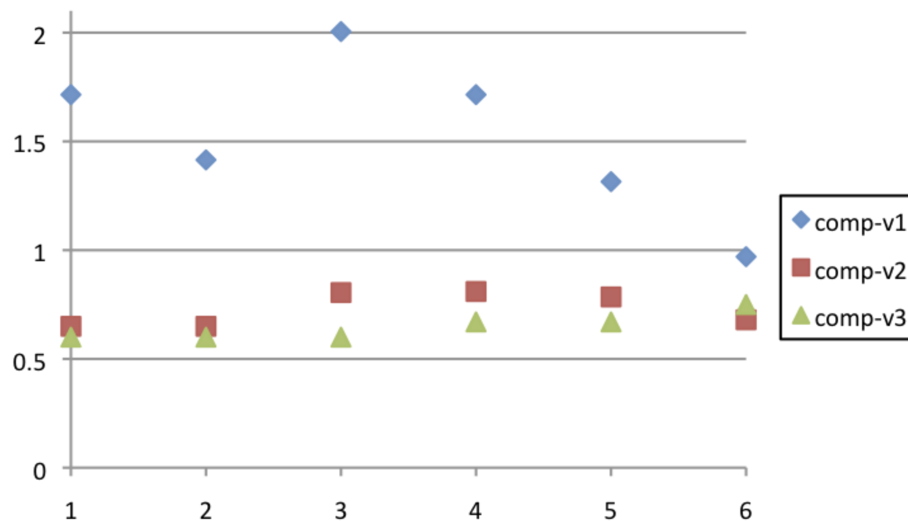

Figure 13.5: Comparison between global indicators

13.3.4 Results

Section 13.3.3 already discussed the relevance and accuracy of the designed measures as well as the limitations of their interpretations regarding quality assess-

ment. This section discusses their usefulness to guide the maintenance process. More precisely, a competent measure should provide support for:

1. The identification of where to apply improvement
2. The identification of the nature of the improvement

Additionally, the measurement results should be consistent with improvement recommendations formulated by the teachers. Regarding this aspect, the recommendations provided during the evaluation in the context of the assignment clearly indicated that the flaws were in fact due to missing functionalities and/or insufficient boundary testing. Neither design nor code would be appropriate to correct such problems.

First version of the MoCQA model

According to these requirements, measures *comp-req-1* and *comp-test-1* show very little potential to assist the maintenance process.

Interpreting the measurement values as a percentage would lead the maintainers mistakenly towards a costly inspection of design and/or code. Indeed, the incorrect interpretation of the measures indicate that the requirements and test plan are complete. This is not consistent with the recommendations formulated by the teachers. Interpreting the measurement values as "anything under 1 is not good" would lead to consider the correct artefacts only for the worst projects (i.e., projects 3 and 6 that possess some use cases without any scenario). The incompleteness of requirements/test plan for the other projects would remain undetected since there is no way to interpret accurately any value above 1.

Even when the artefacts to improve are correctly identified (e.g., requirements of projects 3 and 6) and become candidate to a refinement, measure *comp-req-1* remains unhelpful to identify the nature of the improvement needed. Indeed, since the definition of the measurable entities indicates no relationships between the use cases and use case scenarios, counting blindly the two population sets, quality assessment is not detailed enough to identify the exact use cases that are not covered. The same applies to measure *comp-test-1*.

Second version of the MoCQA model

Measures *comp-req-2* and *comp-test-2* are more useful to guide the maintenance process. First of all, they provide indicators helping us to identify precisely where the problems are. For instance, the development team of project 4 produced a pretty good coverage of their use cases but failed to provide boundary testing for some scenarios. The measurement values point in that direction, targeting the maintenance process towards the improvement of the test coverage instead of towards more support for missing functionalities.

The effort required to apply the measurement plan may be reused to support the maintenance process. Indeed, the definition of measures *comp-req-2* and *comp-test-2* forced the measurer (be it human or a tool) to tag a lot of precise resources of the actual project. For *comp-req-2*, each use case scenario has to be tagged with the related use case. For *comp-test-2*, each test case originating from a precise scenario has to be tagged accordingly. Therefore, in the case of measure *comp-req-2*, the result provides not only the level of completeness but also the identification of functionalities which are more likely missing in the design and/or the code: the functionalities without any defined scenario are more likely to have been completely forgotten by the developers during the development. In the case of measure *comp-test-2*, the measure indicates indirectly which functionalities are more likely to be incorrect due to a lack of testing. Besides, the interpretation is consistent with the recommendation of the teachers (missing functionalities and/or insufficient boundary testing).

13.3.5 Discussion

Regarding our validation process, this case study seems to corroborate the exploitability, adaptability and expressiveness of the framework.

The fact that MoCQA models were designed to fit the investigated context and address the particular problems of the context show the adaptability of the approach in this context. The fact that we were able to provide a better solution thanks to the MoCQA metamodel constructs tends to demonstrate the expressiveness of the approach. It is arguable that the second version of the MoCQA model is better because the designer had the time to rethink the model. But in this case, the case study still increases our confidence in the iterative quality assessment methodology we propose.

Finally, the case study shows that it is theoretically feasible to support the maintenance thanks to the framework, therefore reinforcing our confidence in its exploitability.

13.3.6 Threat to validity

Although the results are encouraging, several threats to validity remain.

The data set of six medium-size projects in an academic and relatively controlled environment is not sufficiently representative of real-world large size evolving software projects.

Besides, the post-mortem application of the framework raises some issues. The exploration of intermediate artefacts was carried out after the delivery of the project (including the maintenance phase) and not really during the development. This remains artificial with respect to real evolving projects as the suggestions of improvement (even answering diagnostics proven to be pertinent) have not been

implemented during the development or maintenance process to provide another data set to compare with.

Finally, these results call for more generalisation. The results have only been validated in this specific context and for the completeness quality characteristic. Other attributes and/or quality characteristics have to be investigated in the future to demonstrate similar results.

Nevertheless, the results should be regarded as a positive reinforcement, regarding our validation process.

Chapter 14

Supporting certification

This chapter describes a practical case study performed in a professional environment that illustrates the adaptability and applicability of the MoCQA framework. It exemplifies the acquisition and design steps of the MoCQA methodology. It provides a context with specific challenges (linked to the certification of software applications) that vary slightly from traditional quality assessment concerns, therefore showing an example of the flexibility of the approach.

14.1 Context

The case study has been performed (and is currently still in progress) at THALES communications Belgium, in the context of the Skywin-SAT project.

Skywin-SAT

The objective of the Smarter Airborne Technologies (SAT) project is to develop new technologies for planes and more intelligent aeronautical systems. The project consists of five axis : two skill centres and three technological axis. The SAT project gathers 16 partners under the coordination of Thales Belgium S.A.[1]

Due to the size and the number of participants involved, the project is subdivided along different themes. The MoCQA framework is currently in use in the certif_2 workpackage. Certif_2 (Certification) has for vocation the certification of on-board critical systems following the software standards RTCA DO-178B (or 178C) and material (RTCA DO-254) defined by the FAA (Federal Aviation Administration, USA) and EUROCAE (Civil European Organization for Avia-

[1] http://www.skywin-sat.be/

tion Equipment). This skill centre is interested more specifically in the aspects of recertification and incremental certification of software product lines[2].

Thales Communications Belgium

Thales Communications Belgium (TCB), the Belgian competence centre of the Thales Group, is internationally recognised for the development and supply of communication systems for the Defence sector and, more generally, for enhanced Security. TCB, a company based in Tubize, Belgium, is the market leader in the national Defense sector. TCB has built up a solid reputation as the developer of a range of cutting-edge technological products and as a systems architect in the field of system integration engineering for critical missions[3].

Among the products developed by TCB, the Multifunctional Airborne Communication System (MACS) is a state-of-the-art Intercommunication System designed to meet the operational communication requirements of aircraft. Thanks to its modularity and distributed architecture, the system can deliver customised solutions for small and large airborne platforms.

Issues

In the context of the Skywin-SAT project, and the certif_2 workpackage, the objectives are to help TCB adopt a selective certification process (compliant with the software standards RTCA DO-178) of embedded communication devices, and more specifically of the MACS.

As a highly configurable application, the MACS may be regarded as a "meta-application" that may be instantiated to provide different complete embedded systems. In this context, each instance of the MACS has to be validated according to the DO-178 standard. However, since only some elements of the configuration vary from an instance to the other, the certification process should not be performed from scratch for every instance, provided that we can accurately pinpoint the elements of the configuration that have been modified, with regard to a reference configuration. Developing a methodology to support this selective certification process is the main objective of the partnerships with Thales Communication Belgium.

14.2 Objectives

Regarding our validation process, the objective is to show that the MoCQA framework is able to adapt to this challenging context and support the selective certification process described above. It also intends to show that the DO-178 standard

[2] http://www.skywin-sat.be/
[3] http://www.thales-communications.be/

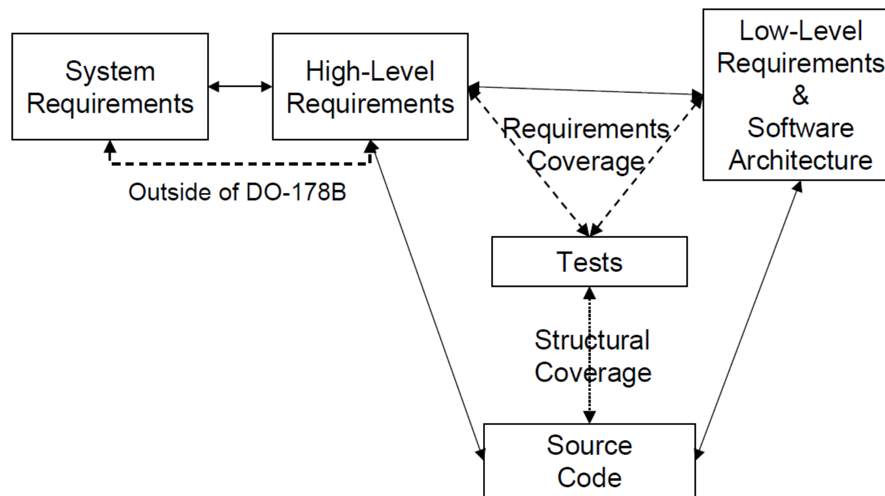

Figure 14.1: Traceability requirements

may be used in the same way other quality models may be exploited regarding the acquisition step.

14.3 Description

As explained in the previous sections, the Multifunctional Airborne Communication System (MACS) provides a system that is highly customisable and can be configured to fit a vast number of contexts (i.e., specific planes). The system may therefore be regarded as a software platform, defined as *"a set of software subsystems and interfaces that form a common structure from which a set of derivative products can be efficiently developed and produced"* in [Meyer and Lehnerd, 1997]. Therefore, methods dedicated to the analysis of variability of Software Product Lines apply to this context. Additionally, the MACS has to comply to the standard RTCA DO-178 in order to prove reliable enough to be embedded in aircraft.

14.3.1 RTCA DO-178b

The standard RTCA DO-178b [RTCA, 1992] describes a framework designed to manage the safety of software used in airborne systems.

As shown in Figure 14.1, the main concern of the DO-178b is the traceability of artefacts during the development. Figure 14.1 shows that the other crucial aspect of the certification process is to ensure the testability (test preparation) and robustness (test execution) of the software application that is reviewed.

In the context of TCB, a reference configuration of the MACS has already been certified according to the expectations of the DO-178b standard.

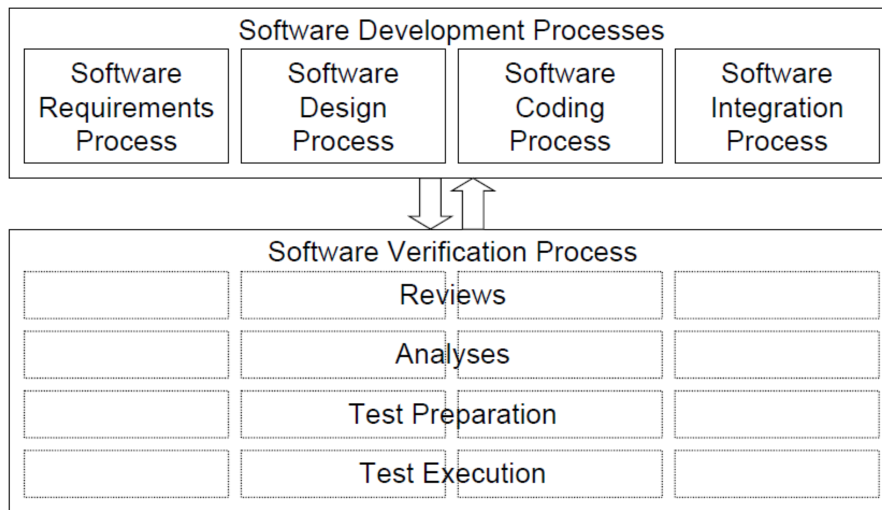

Figure 14.2: Test preparation and test execution

14.3.2 Variability and Software Product Lines

[Pohl et al., 2005] defines variability as *"the commonalities and differences in the applications in terms of requirements, architecture, components, and test artefacts"*. The main mechanism that supports the variability analysis of a platform is the feature models. These models are used to represent all the possible features of a software platform and the constraints that are applicable to them (e.g., exclusivity of features, cardinality, etc.). Based on a feature diagram, it is therefore possible to generate a specific product (or configuration) by selecting the desired features while conforming to the constraints described in the diagram.

Prior to the introduction of the MoCQA approach in the context of TCB, a variability analysis of the MACS was performed by another research partner. A feature diagram was designed and represents the MACS as a platform.

14.3.3 Applying the MoCQA framework

The challenge in the case of the MACS was therefore to reconcile a static quality model defined by the certification standard and a software platform that, by nature, is bound to dynamically provide different systems.

The first step of the case study consisted in eliciting the requirements of the stakeholders at TCB. Based on the DO-178b, it was defined that all the traceability requirements regarding the MACS were already fulfilled. The acquisition step of the MoCQA methodology revealed that the crucial aspect of the DO-178b standard that required investigation was the "testability" and "robustness" quality issues of a specific configuration of the MACS. The challenge raised by these

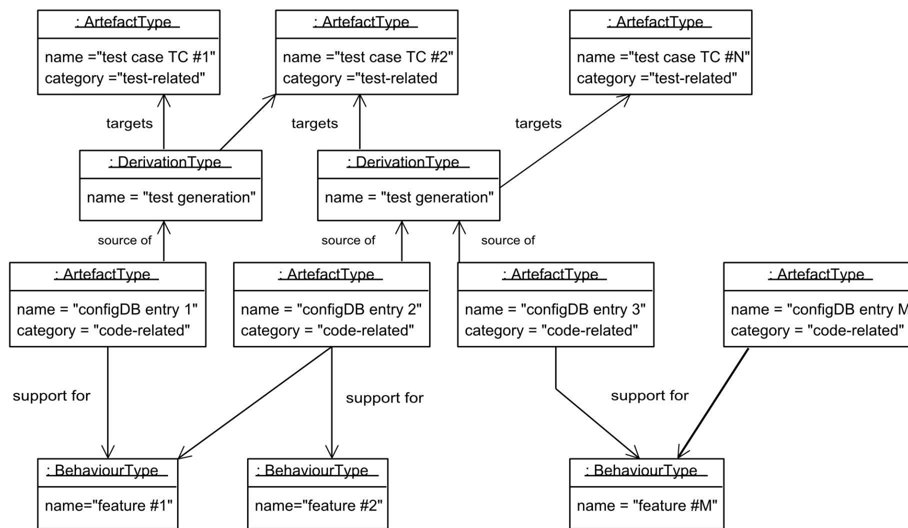

Figure 14.3: Modelling the traceability of test cases

quality issues is to consider the right test cases and demonstrate that all aspects of a specific configuration of the MACS are covered.

During the acquisition step, we also discovered that the main artefact available to guide this process was the configuration database of the MACS. Indeed, TCB developed a database that contains all the possible features a product may display. MoCQA models were thus considered as a way to bridge the gap between the features described by the feature model and the test cases that may target multiple features at once.

Figure 14.3 shows a general view of how the project component may be used to provide such a traceability. As we may see, the artefact types that are used to provide the traceability are the entries of the configuration database. Since a test case may include several features, the test case is also linked to several database entries.

The testability quality issue may therefore be computed on the basis of the number of test cases included and the number of database entries. The robustness quality issue may be assessed through the number of successful test cases over the total number of test cases.

14.3.4 Towards selective certification

The ultimate objective of the project, regarding the certification is to provide tool-support for the generation of a test plan that correspond to a specific "instance" of the MACS. The conjoint use of feature diagrams and MoCQA models has been considered to provide this tool support. Future work will investigate the possibility to tailor a MoCQA model on the basis of a given configuration of the

feature diagram. Due to the way the MoCQA models are designed, it is possible to select a subset of behaviour types based on the subset of features provided by the configuration. Since the project component ensures the traceability between the features/behaviours and the test cases, any modification on the set of features will provide a new subset of test cases to take into account. Comparing the generated test plan to the existing reference certified configuration of the MACS will therefore provide the delta that needs to be tested in order to comply to the DO-178b standard.

14.4 Results

During the course of this case study, the following MoCQA-related activities have been performed:

- Acquisition of the quality requirements based on the DO-178b standard
- Design of a MoCQA model addressing the testability issue for a subset of the MACS
- Early design for a tool that supports the generation of the tests required for the certification

The relevance of the process was assessed through the feedback of the stakeholder and the project coordinator of the Certif_2 workpackage. Although no quantitative data has been collected to evaluate the *level* of satisfaction of the stakeholder, every artefact produced in the context of the MoCQA methodology has been approved.

Based on the artefacts that have been produced and the tasks performed, several points may be noted.

Regarding the integration of the DO-178b, it appears that this standard may be viewed has a quality model and therefore integrated into a MoCQA model. The testability quality issue was included in the MoCQA model to translate the traceability requirements between code and test, while the robustness quality issue translates the actual execution of these tests.

The framework revealed adequate to take the variability of the MACS into account. As explained in Chapter 5, due to their semantics, the behaviour type constructs may be aligned with features of a feature diagram. Therefore, it is possible to provide a MoCQA model with a view that is compatible with feature diagrams.

Although the software context of TCB provides some specific challenges due to its centralised configuration data base, the project modelling constructs were able to address these particular challenges due to the flexibility of artefact types (e.g., data base entry as an artefact type).

14.5 Discussion

The use of the MoCQA framework in the context of TCB tends to show its applicability since the first steps of the methodology have been successfully applied to elicit stakeholders' requirements.

Besides, MoCQA models were used to communicate and validate a long term plan for the achievement of the selective certification.

The fact that the early design of the certification-related tool support integrates XOCQAM documents as a central mechanism also show the flexibility of the approach. Indeed, in this context, the MoCQA framework is used to provide support for a task it had not been initially designed for.

14.6 Threat to validity

A threat to validity in this context is the fact that only one stakeholder participated in the elicitation process. The case study does not show that the MoCQA models are better suited than a traditional requirements elicitation technique.

Another threat to validity is the fact that the long-term tooling relying on the XOCQAM file is not yet developed. The case study therefore does not show that this approach is feasible in practice but only theoretically.

Finally, the case study does not provide quantitative data regarding the criteria that have been assessed.

Chapter 15

Quality Assurance

This chapter describes a practical case study that illustrates the adaptability, exploitability and applicability of the MoCQA framework. This case study took place in a professional environment. The MoCQA framework was used by a quality assurance team of an actual IT department to implement a quality assessment life-cycle. The framework was deployed to maintain and monitor several projects in both production and development states. The MoCQA framework has been applied in the context during one year and half and has now been integrated in the practice of the IT department in question. The application of the framework in this specific context has been addressed in detail in [Hanoteau, 2012].

15.1 Context

The case study took place in the IT Department (D443) of the "Direction Générale opérationnelle de l'Agriculture, des Ressources Naturelles et de l'Environnement (DGARNE)", one of the department of the "Service Public de Wallonie[1] (SPW)", that is, the public administration of the Walloon Region. This IT Department is in charge of about a hundred software products: mainly business applications but also acquired software packages and distributed components. Except for a few isolated cases, no metrics or quantitative assessment of any sort was being used to monitor these products, prior to the introduction of the MoCQA framework. Several internal and specific quality standards were used to guarantee the global quality of projects. In order to fulfil their need of continuous quality assessment and improvement, the quality assurance team of D443 was contacted and the application of the MoCQA framework proposed.

[1] http://spw.wallonie.be/?q=dgo3

15.2 Objectives

The main objective of this case study was to deploy the MoCQA framework in order to ensure the applicability of the methodology. Showing the applicability of the framework requires to ensure that each step of the methodology could be carried out. Besides, it was required to show that the framework actually contributed to the efficiency of the quality assessment process. The success of the application was mainly determined by the reaction of stakeholders (mainly management) to the results of the quality assessment.

Determining the level to which the procedures of the MoCQA methodology were well received by any stakeholder involved in the quality assessment life-cycle was a secondary goal of the case study. The assessment of this goal was conducted through the level of participation of involved stakeholders, their feedback on the assessment and the way the quality team efficiently exploited the framework.

15.3 Description

A preliminary learning phase was required to help the quality assurance team adopt the concepts of the framework. This learning phase was performed through several meetings with the quality assurance team leader, on the basis of the existing MoCQA documentation. In turn, the team leader was in charge of informing his team (constituted of 4 additional members). This learning phase ultimately gave birth to an industrial MoCQA deployment guide (see Section 15.4). Following the learning phase, the quality assurance team of the D443 department started applying the MoCQA framework on a daily basis.

15.3.1 First quality assessment cycle

Acquisition

The first challenge to overcome in this context was the number of possible actors that could have been selected as stakeholders. The IT department in which the case study took place counts 70 agents. They manage a pool of software applications used by a total of 2400 users in the DGARNE. This wealth of possible stakeholders lead to a selection of 5 stakeholders. This selection was based on the availability and role of the actors. During this first step, the relevant classification was identified as a distinction between applicative stakeholders (i.e., any stakeholder that has to act on the software applications, regardless of his specific role in the process) and management. Out of the 5 individual stakeholders, 3 were coming from management and 2 from the applicative stakeholders (i.e., team leaders). The acquisition step was performed by the quality assurance team leader, through a round of individual interviews with each stakeholder. This pro-

cess was formalised as a series of internal reports complying to the template in use in the public administration.

This initial round of interviews lead to the elicitation of 26 quality goals/requirements. They were classified, organised and prioritised with the help of the head management of the D443 IT department, who may therefore be considered as an additional stakeholder. The priority was given to the "reliability" requirement for the first quality assessment cycle.

MoCQA model Design

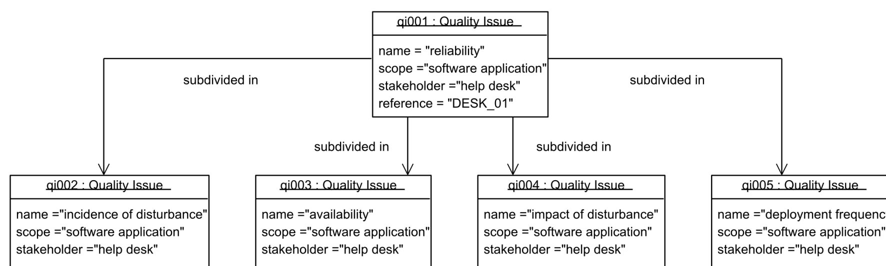

Figure 15.1: Example of quality issues expressed during the case study

Based on the structured list of quality requirements, the hierarchy of quality issues shown in Figure 15.1 was designed. As we may see, although the "reliability" quality factor may appear to originate from the ISO/IEC 9126 quality model, it is fact inherited from the internal standard of the organisation. Therefore it is decomposed a the following series of specific sub-issues:

- Incidence of disturbance
- Availability (of the software application)
- Impact of the disturbance
- Deployment frequency

These quality issues encompass all the relevant reliability aspects of a software application in production in the environment of the case study. The first quality issue is concerned by the frequency of unexpected behaviours from the software application. The second quality issue complements the first and is concerned by the overall availability of the application over time. The third issue intends to measure the criticality of the disturbances. Finally, the deployment frequency quality issue intends to provide a sense of the number of times the system has to be modified and re-deployed, following a major disturbance. Note that the name, although non conventional are inherited from the internal standards but could be aligned with other standards (e.g., the availability in this context may be aligned with the fault-tolerance characteristic of the ISO/IEC quality model).

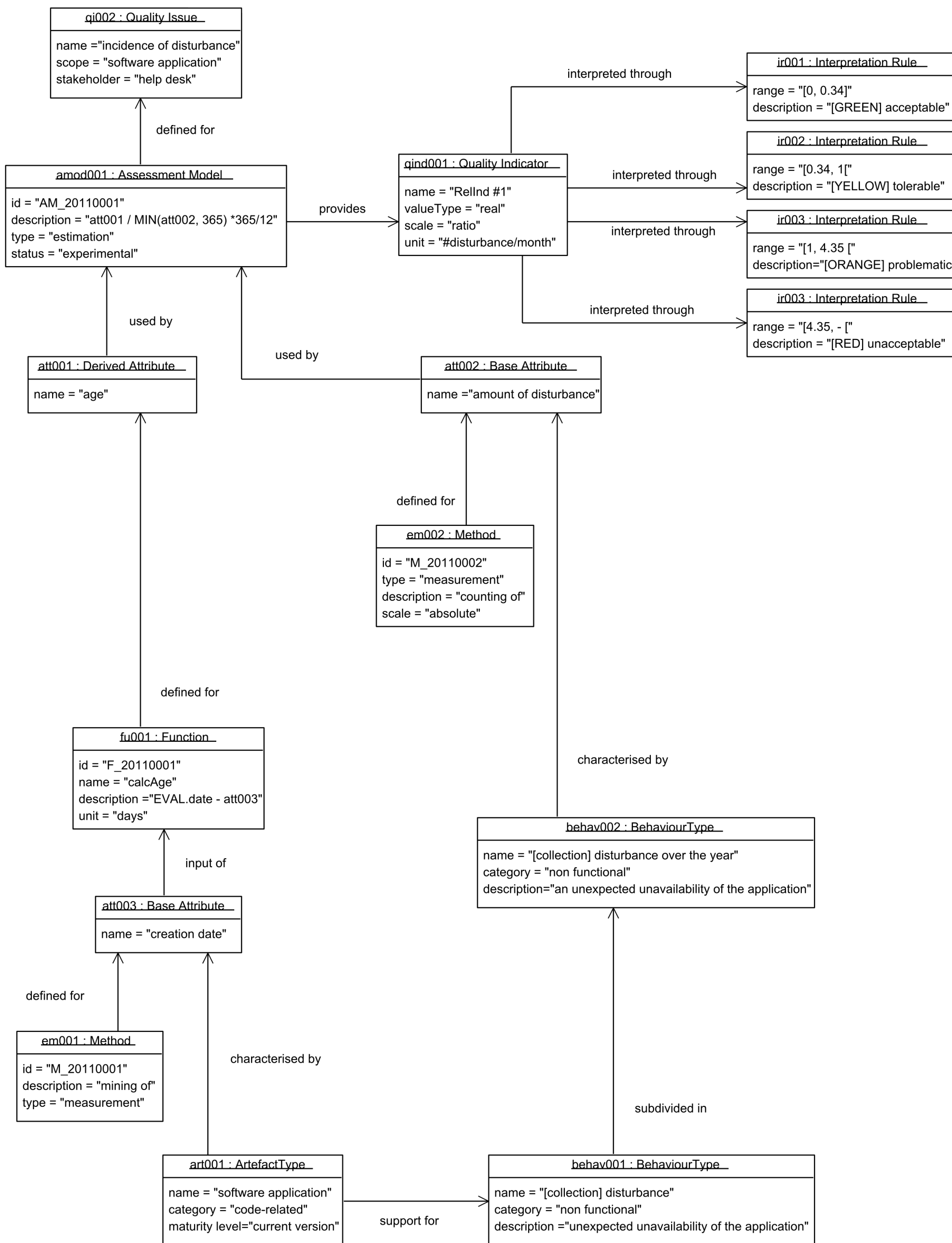

Figure 15.2: Example of MoCQA model designed during the case study

According to the prioritisation performed in the acquisition step, the first quality issue that has been addressed is the "incidence of disturbance" quality issue. Figure 15.2 provides the MoCQA model for this issue. As shown in the model, the quality assessment described mainly relies on behaviours. Compared to previous case studies, the design step of the MoCQA model provided us with

a new challenge. This challenge related to the introduction of temporal constraints. This quality issue requires the introduction of a number of disturbances per period of time. Although it was not supported by any specific procedure or modelling construct of the framework, MoCQA models revealed flexible enough to integrate this information almost seamlessly. The problem was solved through the introduction of an "EvalDate" variable in the description of the functions and assessment models. This variable represents the current date at the time the assessment is performed. Similarly, the age of the software product may be modelled as a derived attribute based on its original deployment, measured as a base attribute.

The other quality issues were also modelled during this first design step, relying on similar measurement methods.

Measurement plan tailoring

The operationalisation of the MoCQA model mainly consisted in providing measurement procedures. For this MoCQA model, the measurement methods were identified as "repository-mining" methods. Therefore, the two repositories were identified (one for the acquisition of the original deployment date and one for the report of disturbance) and specific SQL queries were designed for each of the measurement methods. All other computations were performed manually, although the functions and assessment models were formalised in C#, in prevision of a future automation of the process. The data collection was planned using spreadsheets.

Assessment and Exploitation

Based on the operational customised quality assessment model, the members of the quality assurance team were able to apply the model to the 56 software projects selected during the acquisition step.

The first exploitation step was performed with the management stakeholders. They were explained the quality assessment process on the basis of the MoCQA model. The quality assessment process was agreed upon and the assessment results analysed. The decisions taken on the basis of this first quality assessment cycle were mainly related to the continuation of the quality assessment life-cycle. Modifications in the interpretation rules, including the addition of specific recommendations regarding required corrective actions. However, the assessment results were perceived positively by the stakeholders as reinforcing pre-existent intuitions on several software applications of the pool.

15.3.2 Continuation of the quality assessment life-cycle

The next quality assessment cycles focused on the refinement of the MoCQA model in order to support corrective actions. During the second quality assessment cycle, exploitation occurred with the contribution of applicative stakeholders.

New quality issues were added with each new quality assessment cycle. At the end of the case study, 14 quality issues were monitored with the support of the MoCQA framework.

15.4 Results

During the one year and a half lifespan of the case study, each step of the MoCQA methodology has been applied several times. The quality assurance team leader reported his progress and results to the management of the D443 department on a regular basis, in order to define if the course of the project was considered satisfying and should be continued. Subsequently, the quality assurance team leader provided us with reports on the events. Details on these reports are available in [Hanoteau, 2012].

Similarly to the Thales case study, the relevance of the MoCQA framework has therefore been assessed through the feedback of the stakeholders and the quality assurance team leader. No quantitative data has been collected to evaluate the *level* of satisfaction of the stakeholders but the application of the framework was not discarded by the management at any point.

Based on the reports provided by the quality assurance team leader, several points may be noted.

First, the iterative and incremental aspects of the methodology have been accepted and applied. Although no quality indicator or measurement/estimation method had to be deprecated during the course of the study, the apparition of new quality requirements leading to new quality indicators occurred and was supported by the methodology.

Regarding the design of MoCQA models, the constructs provided by the quality assessment metamodel were all used at some point of the process. The quality assurance team was not confronted to a quality related problem that they could not model and respected the semantics and syntactic rules of the quality assessment metamodel.

The assessment performed on the basis of the MoCQA methodology was considered relevant according to both management and the quality assurance team. The quality indicators defined during the quality assessment life-cycle of the study were accurate in their support for refinement. The quality assessment performed based on the MoCQA models led to the inspection and maintenance of several software systems and to the refactoring of the help desk supporting repository.

The deployment of the MoCQA framework in the context of the IT department D443 also allowed to determine how manageable the assessment methodology is in terms of costs. Due to organisational requirements, costs where recorded and communicated to us for the first quality assessment cycle of the two first quality issues investigated by the quality assurance team. For those two quality assessment cycles, the costs were estimated to an average of 10 man-days[2] per cycle. These costs were reviewed by the quality assurance team and the management and considered acceptable (i.e., not inducing an unacceptable overhead). The main overhead was identified as the initial learning phase, evaluated to 14 man-days.

Finally, the framework has been chosen as a candidate to be integrated in the D444 department as a full-fledged quality assessment support for the quality assurance team.

15.5 Discussion

Regarding our validation process, this case study reinforces the positive perspective on the adaptability, exploitability and applicability of the MoCQA framework.

The adaptability is shown by the fact that the framework helped model the specific quality requirements in the studied environment, whereas the exploitability is shown by the fact that the assessment results were well received and led to actual actions carried out in the studied environment.

Regarding the applicability, the framework was reported to be used without any hindrance by a third party (i.e., the quality assurance team), past the learning phase. The cost estimation, although not providing general results, tends to show that the overhead induced by the MoCQA methodology is not a stumbling block, in comparison to the benefits it provides.

Additionally, the reports provided us with several observations about the use of the MoCQA framework and Software Quality in general. The remainder of this section reports and elaborates on these observations.

Industrial deployment guide and language issues

The first observation made in the context of the D443 department was the initial difficulty to apprehend the concepts of the quality assessment metamodel. As it turned out, this difficulty was not an intrinsic issue of the framework. The issue was rather related to the terminology used in the framework and in the software quality literature.

[2] In the context of the D443 department, a man-day is assimilated to 7.6 hours of work for 1 employee

During the initial learning phase of the case study, several interactions with the quality assurance team of the IT department revealed that the academic terminology tended to obfuscate the meaning of otherwise easily understandable concepts. Through regular exchanges with the quality assurance team leader, the concepts of the framework were adapted to the terminology of the D443 department.

Another terminology issue was discovered during the exchanges with the quality assurance team. This issue regards the use of "software project" to define the level of the quality assessment performed with the framework. In the context of the IT department D443, the term "project" is used according to the definition provided in [PMI, 2004] (i.e.,*"a temporary endeavour undertaken to create a unique product or service"*. This process oriented perception of "project" is compliant with the second part of the definition provided by the CMMI framework: *"A project has an intended beginning (i.e., project startup) and end. Projects typically operate according to a plan. Such a plan is frequently documented and specifies what is to be delivered or implemented, the resources and funds to be used, the work to be done, and a schedule for doing the work. A project can be composed of projects"*. However it led to initial conflicting understandings of the scope of the framework. In the absence of a better word to specify the scope of quality assessment modelling, the term software project was kept but the case study showed that it is necessary to emphasise the definition of software project provided in Chapter 4.

Besides, one of the first activity required during the case study was the adaptation of the MoCQA framework to the working language of the Walloon public administration (i.e., French). The quality assessment metamodel and all the required documentation was therefore translated into French during the case study. This process partially helped the learning process for the quality assurance team.

In order to circumvent those initial hiccups, an industrial MoCQA framework deployment guide was designed in collaboration with the quality assurance team leader of the IT department. This guide, described in [Hanoteau, 2012] and only available in French so far, intends to formalise the terminology adaptations. It therefore provides a more practical perspective on the framework and is supposed to be generic enough to adapt to any company. It provides a comprehensive description of the quality assessment metamodel and its constructs. It provides a practical description of the MoCQA methodology and of the quality assessment life-cycle implemented by the framework. Finally, it provides several examples of MoCQA models and explores the modelling conventions designed during the case study.

15.5.1 Impact of quality indicators

During the course of the case study, we also had the opportunity to observe the impact of the introduction of formalised quality indicators in the context. At the end of the first quality assessment cycle, the assessment results provided stakeholders with unsurprising conclusions. For the most part, the problems reported by the quality assessment model were known or sensed to some level by the management stakeholders. However, the introduction of quality indicators and the rationale behind these values helped reinforce the motivation to take actions in order to solve the problems. Although the indicators introduced in the first quality assessment cycle were not very specific or refined, their impact was already important. Moreover, the notion of iterative quality assessment life-cycle guarantees that problems reported at the beginning of the process will be reported again recurrently. This iterative mechanism acts as a reminder of known problems.

The quality assessment process itself may also lead to interesting conclusions that impact the environment. Since the exploitation step analyses both the assessment results and assessment process, it is possible to report valuable information while trying to improve the quality assessment process. For instance, the end of the first quality assessment cycle showed that the collection of data was hampered by the lack of a centralised repository to find the necessary data (i.e., mainly the reported disturbances). Each software application was managed separately. The exploitation step led to the decision to centralise the information on the various software applications.

Regarding the interpretation of the quality indicators, the iterative methodology was also well received. The caveat with indicators in general is to avoid interpreting them without a critical view on what reality they encompass. The fact that the framework allows for a critical revision of quality indicators (e.g., modify the threshold of over-demanding quality indicators) and provides the formalised rationale behind the quality indicator was beneficial for the fine-tuning of quality assessment over time.

15.5.2 Human aspects

During the course of the case study, we also had the opportunity to confront the MoCQA framework to the perception of the various stakeholders. Some reluctance or scepticism towards the introduction of a formalised quality assessment framework appeared during the first and second quality assessment cycles. This circumspection took different forms depending on the type of stakeholders.

The management mainly worried about the return on investment of the application of the MoCQA methodology. The concern was thus the amount of time

and effort the deployment of the framework would require. The conclusion of the first quality assessment cycle provided reassuring answers to this concern.

The applicative stakeholders were more concerned by the quality indicators themselves, raising the issue that the quality indicators may not reflect the truth of the applications they were responsible for. As explained before, this reaction is not surprising since individuals tend to dislike the notion of quality control. During the second exploitation phase, explaining to them the fact that taking into account their feedback on the results and interpretation was part of the process helped solve the issue.

A transversal issue regarding the deployment of the framework was also raised during the first quality assessment cycle. This issue was related to the perceived "subjectivity" of the quality assessment process. The choice of reliability as a first quality issue was questioned by other stakeholders. The same occurred with the way quality issues were assessed. Following the regular MoCQA methodology, this concern was integrated into the decision-making process regarding the quality assessment process. Therefore, the input of stakeholders that were not concerned by the reliability was used to decide which quality issue should be investigated next. The assessment process for reliability was maintained after exchanges between the quality assurance team and the aforementioned stakeholders

The participative nature of the framework was therefore well received by the various stakeholders and helped cope with their concerns.

15.5.3 Stakeholder classification

As explained previously, a light classification was proposed in the context of IT department D443. This dichotomous categorisation turned out to be sufficient during the course of the case study. The two categories of stakeholders clearly elicited different goals and, as seen in the previous section, different worries regarding the quality assessment process. Note that the dual classification is also sufficient due to the general quality approach adopted in this specific context. Since the indicators are mainly high level and are attached to global products, providing a more refined typology of stakeholders (e.g., designers, etc.) is not relevant since nothing in the designed MoCQA models is targeting specific types of stakeholders.

The dual classification management/applicative stakeholders also reveals an inherent specificity. As a matter of fact, management stakeholders may also be perceived as "generic" stakeholders. Their quality requirements are the same for each system investigated (and therefore more high-level). On the contrary, applicative stakeholders may have needs that slightly vary from a software application to the next.

Another interesting observation regarding this dichotomy is the fact that the way measurement and assessment results are introduced to the type of stakeholder

varies slightly. Basically, we distinguished two tendencies.

- Managerial stakeholders are more prone to react positively to dashboards. Although the presence of the MoCQA model itself is reassuring, the outcome management stakeholders are expecting is a set of indicators.
- Applicative stakeholders are more prone to react negatively to dashboards. Providing a set of values to the individuals that actually act on the software applications raises concerns on the origin of the values and how they were computed. In that case, the support of the MoCQA model helps provide a good understanding of the rationale behind the indicators in a format that is familiar to the applicative stakeholders (i.e., models).

15.5.4 Target of the assessment

As expected, an important aspect of the deployment of a quality assessment plan was to communicate on the target of the assessment. The key to a successful assessment is to prevent individuals from feeling assessed themselves. The hybrid point of view of the framework (i.e., product/process) helped reassuring the development teams on this point.

First, the availability of the MoCQA model provides a transparent way to clearly define the goals of the assessment. Through the consultation of the model, each member of the department (even if they are not listed as stakeholders) may understand the process. MoCQA models provide many constructs but clearly none of them is designed to assess individuals. Therefore, the quality assessment process was well received in the context of the study.

On the other hand, the concept of derivation type helps manage the most sensitive aspect of the process. Whereas measuring a process may be perceived as a way to point to some individual mistakes, derivation types provide an abstract concept that removes completely the notion of individual assessment. Derivation types were used during the assessment of the second main quality issue (i.e., "compliance life-cycle"). Although the quality issue clearly relied on the assessment of individuals' performances or skills, the fact that derivations made the implication of these individual abstract helped increase the overall acceptance of the process.

15.5.5 Availability of results

The question of how the availability of quality assessment results would be managed came early in the deployment of the framework. The issue with vastly available quality assessment results is that people tend to compare results among them. Although, this could be regarded as a way to motivate people, this induced many concerns in the context of the study.

The classification of stakeholders helped manage the availability of results. It was decided that only management (i.e., generic) stakeholders would have access to the complete data set. The applicative stakeholders were provided with data related only to the software application they were involved in.

15.5.6 Support from the management

The case study also showed that quality assessment must be management-driven in order to be productive. Although the framework provides many elements to counter the reluctance of scepticism from the development team (i.e., participative and iterative methodology), the framework must be applied with the full support of the management. During the course of the case study, the support from the management helped the quality assurance team motivate and decide the development team to take part in the quality assessment and improvement processes. This observation reinforces the considerations provided by [Westfall and Road, 2005]. The fact that each quality indicator is defined with a given purpose (originating from the management) helps reinforce the perception that quality assessment is a useful process. Additionally, the management has to clearly support the viewpoint described in previous sections (i.e., the guiding perspective of quality assessment). The fact that the management supported the deployment of a framework that relies on this “guiding over control” philosophy greatly helped reassuring the applicative stakeholders in the studied environment.

15.6 Threat to validity

Although the results of this case study are positive, they once again only apply to this specific context. The cost estimation cannot be generalised at this point since this aspect is highly sensitive to the context of use and the complexity of the designed MoCQA model. The support provided by the management was crucial to the success of this deployment. Besides the views of the quality assurance team were already close to the underlying concepts of the framework. In other words, the environment of the case study was perfectly suited to introduce the MoCQA framework. Results therefore call for generalisation in other contexts.

Another issue of the case study relates to the quality issues investigated during its course. The MoCQA models designed during this study do not provide really robust project components. For instance, the MoCQA model shown in Section 15.3 is not suitable to perform a root-cause analysis. This aspects is mainly due to the fact that the quality assurance team had to tackle many software applications. In that context, it was not possible to define more precise project components. This could be solved by differentiating the models for each software application. However, the overhead induced by such an approach could influence

the costs and effort drastically. Still, the flexibility of the framework provides the opportunity to specialise MoCQA models only when required.

Finally, the case study does not provide quantitative data regarding the criteria that have been assessed.

Part IV

Closing comments

Chapter 16

Discussion

Contribution, review and perspectives

This chapter recapitulates the main contribution of this thesis (Section 16.1) and discusses its potential to improve the way software quality is envisioned and managed (Section 16.2). Current limitations and improvement points are also addressed in Section 16.3. The chapter ends with an overview of research perspectives and future works motivated by our research work (Section 16.4).

16.1 Contribution

During the course of this research work, we developed a theoretical approach to quality assessment in an attempt to provide a better support for the definition of software quality goals and the monitoring of their level of achievement in the context of software development. Based on a review of the lasting impediments that prevent software quality assessment to be exploited at its full potential, we identified three core notions that could be beneficial to our proposed approach:

- **Model-driven quality assessment** (i.e., using a *quality assessment model* to support the elicitation of quality requirements, to plan the quality assessment activities and to facilitate the communication, the analysis and the exploitation of the measurement results.)
- **Explicit and integrated quality assessment modelling** (i.e., making the quality assessment models *operational* and *customised* in order to ensure that the information recorded in the models is centralised, fits the specific context in which the software development process occurs and conveys enough elements to prove useful to each involved stakeholder.)

- **Dedicated quality assessment life-cycle** (i.e., manage the design, refinements and evolution of *operational customised quality assessment models* through a distinct life-cycle that follows closely the sequence of events inherent to the software development life-cycle.)

The introduction of these three notions aims to integrate the quality assessment process further into the software engineering process itself. By allowing a continuous back-and-forth exchange between the parallel software development and quality assessment life-cycles, by means of quality assessment models, the theoretical approach aims to improve the quality assessment process regarding the following aspects:

- **Support**: help plan and adjust the quality assessment process throughout the entire software development life-cycle, from early stages and on to maintenance/evolution.
- **Relevance**: providing a quality assessment that fits the specific context in which it is performed and avoids wasting time on pointless measurement/assessment efforts.
- **Communication**: ensuring that all stakeholders share the same view on the quality goals for the project and accept the quality assessment process, as well as helping each of them act individually so that the overall team converges towards these goals.
- **Awareness**: providing support to detect the flaws in software measurement methods and mistakes in the way quality is envisioned for the project, as well as improving the common understanding of quality concerns among the various stakeholders.

Relying on founding principles inherited from other fields of Software Engineering and on documented good practices of Software Quality and Measurement, we successfully implemented this approach to quality assessment into an concrete and operational framework.

This Model-Centric Quality Assessment (MoCQA) framework defines an *iterative and incremental* assessment methodology that focuses on a *goal-driven definition of measures*. This methodology relies extensively on the *involvement of the stakeholders* and let the stakeholders steadily *construct* a common mental model of the quality aspects at stakes for the software development project. It emphasises a proactive treatment of the *human aspects* involved in the measurement program (i.e., it envisions measurement as a guiding activity instead of a control mechanism).

The methodology allows the exploitation of operational customised quality assessment models (or MoCQA models) through a dedicated quality assessment metamodel. This quality assessment metamodel guarantees the *integration of declarative and analytical approaches* in MoCQA models and let these models

adopt an *ecosystemic viewpoint* on software quality. Additionally, MoCQA models are supported by two *domain-specific languages* that increase the usability and efficiency of the framework, while granting a better *reusability* of the components of the models.

In consequence, the MoCQA framework has the potential to provide the necessary support for the integration of various quantitative quality assessment methods (both existing ones and customised ones) into any type of development and maintenance life-cycles in a meaningful (i.e., useful for all stakeholders), self-aware (i.e., allowing a critical review of the measurement results) and flexible (i.e., allowing to adapt easily to any type of development and maintenance life-cycle) way.

16.2 Review

During the course of this research, we applied the MoCQA framework to various case studies, both theoretical and empirical. In order to assess how the framework concretely leverage the potential of our theoretical approach, we defined a series of research questions to guide our effort (see Chapter 10 for more details). This series of questions were classified in two categories, one focusing on the usability, the other addressing the effectiveness of the framework.

The first category was formalised as follows:

[**RQ1**] Is the MoCQA framework usable?

[**RQ1a**] Is the model-driven quality assessment methodology defined by the framework applicable in practice without negative impacts on the rest of the development process?

[**RQ1b**] Is the MoCQA framework accepted and adopted by all involved stakeholders?

[**RQ1c**] Are MoCQA models apt to model the necessary quality assessment information and support the communication of this information between stakeholders?

Regarding **RQ1a**, Chapter 14 and Chapter 15 showed that following the iterative model-driven methodology proposed by the framework is feasible in professional environments. Additionally, during the course of the case study described in Chapter 15, we did not encounter any major issue regarding the maintenance and evolution process that was performed in parallel to the quality assessment life-cycle. Similarly, this case study reports that the MoCQA framework, in that context, did not cause an unacceptable increase in cost, time or effort.

During the course of the case study described in Chapter 15, the majority of the stakeholders did not display any major opposition or reluctance to integrate

the quality assessment life-cycle. The participative nature of the methodology was globally accepted and even succeeded in tempering some issues regarding the choice of the quality indicators. Although these results are less conclusive, due to the relatively controlled context, the quality assessment metamodel was also adopted by the students without any major hindrance (as shown in Chapter 13). The answer to **RQ1b** therefore tends to be affirmative.

Similarly, **RQ1c** seems to be corroborated by the case studies. MoCQA models were used as the main mechanism to justify and exchange information on the quality assessment process in case studies reported in both Chapter 14 and Chapter 15 without problematic miscommunication.

Ultimately, the case studies we performed tend to show that the MoCQA framework is usable in a professional environment. Past the inevitable learning-curve, it does not hamper the course of the software development and was accepted and apt to support the quality assessment process in the contexts we investigated.

The second category of research questions was formalised as follows:

[**RQ2**] Is the MoCQA framework effective?

[**RQ2a**] Does explicit and integrated quality assessment modelling succeed in accurately modelling the specific quality requirements for a given context?

[**RQ2b**] Does the quality assessment methodology helps provide a targeted assessment that meets the specific quality requirements?

[**RQ2c**] Does the iterative use of MoCQA models help plan and adjust the quality assessment process throughout the software life-cycle, from early stages to maintenance and evolution.

[**RQ2d**] Do MoCQA models help detect the flaws in the quality assessment process that is performed?

[**RQ2e**] Do MoCQA models help identify the corrective action that have to be performed in order to improve the level of satisfaction of the quality goals?

Regarding **RQ2a**, the case studies showed transversally that it is possible to design a MoCQA model that addresses adequately each specific problem we encountered in the investigated contexts. Chapter 11 to 13 provided theoretical responses to specific problems occurring at early stages of the development, while Chapter 14 and 15 demonstrated the ability of MoCQA models to fit actual ongoing software development life-cycles.

In the case study of Chapter 15, the framework was used to provide a quality assessment process that met the expectation of all involved stakeholders. Chapter 12 and 13 also showed that it is theoretically feasible to provide a focused

quality assessment that actually helps the development team to avoid further efforts. The answer to **RQ2b** thus tends to be positive as well.

Regarding **RQ2c**, Chapter 11 to 13 provided theoretical responses to specific problems occurring at early stages of the development, while Chapter 14 and 15 demonstrated the ability of MoCQA models to fit actual ongoing software development life-cycles.

RQ2d and **RQ2e** found a reasonable number of positive hints in the case studies we performed as well. Chapter 13 showed that is was possible to analyse a MoCQA model and identify the mistakes that were made regarding the quality assessment process in relatively controlled and safe contexts. Chapter 12 showed the theoretical possibility of MoCQA models to support the decision-making process regarding software architecture. Additionally, during the course of the case study described in Chapter 15, MoCQA models were consistently used to identify points of actions for the monitored software projects.

In conclusion, the case studies we performed provide enough positive reinforcement regarding the effectiveness of the MoCQA framework. In the contexts we investigated, the framework was able to leverage the expected benefits of the theoretical approach we developed.

16.3 Limitations

In spite of the overall positive outcome of our validation process, the MoCQA framework and its underlying theoretical foundations still have to be thoroughly investigated in order to gain in maturity and be regarded as a viable approach for the industry. This maturation process has to cope with the limitations detailed in the remainder of this section.

The first limitation the MoCQA framework has to overcome is the lack of generalisation of the results presented in this dissertation. Although the research questions regarding the approach have been answered in specific contexts, the usability and effectiveness of the approach has to be demonstrated in other contexts. So far, the MoCQA approach has been applied successfully in two separate professional environments but, at this stage, we cannot guarantee that every context will allow a suitable integration of the approach. Only through repeated empirical studies in various contexts will the approach collect enough evidence of its advantages, or reveal other shortcomings that the approach needs to overcome. Additionally, the industrial case studies performed during this research mainly focused on the feasibility (i.e., *usability* and *effectiveness*) of quality assessment processes relying on the MoCQA framework. Future case studies should therefore investigate the *efficiency* of the approach (i.e., whether or not the approach actually increase the productivity and the cost-effectiveness of quality assessment).

Another limitation of the validation process performed so far is the fact that the approach has not been applied in an actual software development process conducted from scratch. Although the proposed approach has been designed in order to support the integration of quality assessment from the very beginning of the software development process and continuously as this process unfolds, not enough opportunities to validate the effectiveness of this integration have been encountered during the course of this research. Without empirical studies aiming to confirm the effectiveness of the approach as a continued quality assessment framework, we still cannot ensure its viability in the industry (i.e., the fact that the methodology is applicable and that the early indicators may be refined in an efficient way without hampering the development process) past the theoretical feasibility that has been demonstrated.

Besides, the approach does not solve (nor does it aim to solve) the problem of the selection or definition of adequate and valid measures. This topic has been (and still is) studied extensively and is intimately linked to the maturation process of Software Measurement. While multiple approaches (such as GQM-MEDEA or SMML) may be applied in conjunction with the MoCQA framework to strengthen the measures selection and definition process, the framework itself only aims to structure these activities from a methodological and pragmatic point of view, allowing the quality assurance team to document this process and correct it throughout the quality assessment cycles.

Another limitation of the approach is the fact that the indicators defined in Section 8.3.2 in order to provide concrete means to detect the possible flaws in the design of MoCQA models have not been theoretically nor empirically validated. Even though they have been defined in a consistent way, the fact that any measure proposed in software engineering should undergo a careful process of theoretical and empirical validation cannot be ignored, especially regarding the nature of this research work. Although not yet crucial at this point, this validation process will gain in relevance as the approach matures and, will have to be considered in the future.

Finally, although we provided some demonstration of the expressiveness of the approach regarding the integration of existing quality models, we cannot guarantee that any possible quality model may be integrated in a MoCQA model as-is. Provided that a quality model displays a hierarchical structure defining high-level factors refined progressively towards a series of attribute-like criteria, the quality assessment metamodel theoretically provides the necessary support for the alignment and integration process. However, a more systematic validation process, as the one proposed in [Klaes et al., 2010] could provide more irrefutable evidence. In the same way, quality models and quality assessment approaches based on non-hierarchical structures (e.g., using Bayesian networks, such as [Vaucher, 2010]) have not been addressed during the course of this research. These ap-

proaches offer several advantages (e.g., the opportunity to include probabilistic impacts of quality factors on each other) and should be investigated in the context of MoCQA. This kind of non-hierarchical relationships between factors may be modelled as assessment models. However, the feasibility of such an integration should still be validated on concrete cases in order to guarantee the expressiveness of the quality assessment metamodel. In the future, investigations on how they can be used in conjunction with the MoCQA framework should therefore be carried out.

16.4 Perspectives

The research described in this dissertation attempts to provide an original view on software quality and introduces notions that aim to change the way quality assessment is performed and perceived. As such, it opens multiple research perspectives, both regarding the approach itself but also in regards to other software engineering fields and research topics. The remainder of this section elaborates on some of these opportunities.

More case studies in industrial context

As explained in the previous section, the MoCQA framework still needs to be used in actual professional environments in order to gain in reliability and maturity. The more the framework will be used concretely, the more the confidence in the approach will increase. Many of the theoretical options offered by the approach have still not been field-tested.

First, during the course of this research, we did not have the opportunity to actually apply the framework in the context of a project starting from scratch. This limitation will be addressed in the close future, thanks to a collaboration with the Centre of Excellence in Information and Communication Technologies (CETIC)[1]. In the context of a FEDER funded research project promoting the collaboration between industrial and academic partners, the application of the MoCQA framework has been considered by the CETIC to support the set up of a quality assessment plan for a software project starting at the *Office de la Naissance et de l'Enfance (ONE)*[2]. The software project is still in its inception phase and the MoCQA approach is currently used to formalise the quality requirements already collected by the CETIC. This collaboration will provide the opportunity to test the usability of the framework earlier in the development process. This will also help us test the guiding potential of the approach for the development team.

[1] http://www.cetic.be/index-en.php
[2] http://www.one.be/

Another aspect of the framework that has not been put to use in an professional context so far is its potential to detect the semantic inconsistencies of the quality assessment performed base on the indicator defined in Chapter 8. The case studies described in Chapter 14 and 15 are still ongoing and have already accumulated an important amount of MoCQA-related material. They should therefore provide suitable contexts to test this potential.

However, finding more opportunities to field-test the approach is still required. In order to convince more partners to deploy the MoCQA framework, it is essential to understand which factors may slow its adoption rate. The main reluctance regarding the application of an experimental quality assessment framework is its experimental status. Software quality is a delicate subject matter for which efficiency is of the uttermost importance. Potential users may regard a new quality assessment framework as a risk to produce erroneous results leading to costly mistakes. The second factor that may discourage potential users to adopt the framework is the learning-curve of the approach as well as the efforts required to apply it.

Efforts required to counter these factors regard the way the MoCQA framework is introduced to the partners. Putting the focus on the integrative nature of the approach should help convince the potential users that MoCQA is not just another quality framework but builds on the existing software quality body of knowledge to integrate suitable quality assessment methods. Similarly, it is important to emphasise the fact that the model-driven nature of the approach has the potential to help spare time and effort, due to the existing set of tools that exist to support the approach, as well as the fact that MoCQA models offer a more concrete and reusable perspective on quality assessment.

Finally, future industrial case studies should focus on obtaining more quantitative results regarding the validation process. Obtaining data regarding the productivity and cost in an industrial context is a complicated task. Each project is unique and therefore, the comparison of quality assessment costs and efforts across projects is not relevant. Future studies could however address the problem of quantitative validation thanks to more structured approaches such as satisfaction surveys to determine the level of satisfaction of the stakeholders.

Empirical studies

As for now, more empirical evidence is still required to demonstrate that the MoCQA quality assessment metamodel offers a better definition of measures due to its specific perspective on the measurable entities. Although the exploratory study addressing the use of completeness quality issues in the maintenance process of medium-sized student projects (see Chapter 13) provides encouraging results, these results have only been validated in a specific context (small projects, only

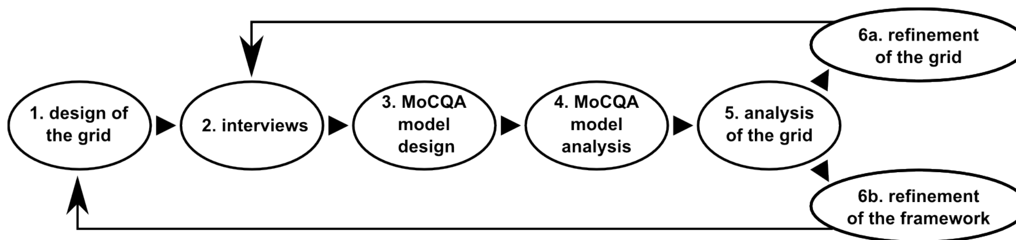

Figure 16.1: Collaborative and iterative validation/refinement methodology

completeness indicators, etc.) and call for both generalisation and more statistical evidence.

In order to generalise the results, future studies could focus on other quality factors. The reliability characteristic should be investigated since its scope is close to the completeness study (i.e., the use of test cases to evaluate the maturity/fault tolerance instead of the completeness). The efficiency characteristic is also a good candidate to test the MoCQA viewpoint since the capability to trace the impact of specific artefacts on external attributes (such as response time, memory use, etc.) should prove valuable for software maintenance.

To acquire more statistically relevant results, future studies should focus on larger sets of projects. A possibility would be to rely on smaller individual student projects with more focused requirements. Although these projects would be smaller, their increased availability would help us apply more sophisticated statistical tests. Open-source software projects should be considered too, since the availability of successive software versions would allow us to monitor the evolution of the measurement values and analyse their relationships with the changes made by the development team.

Refinement of the acquisition step

Among the various solutions proposed in Chapter 6 in order to provide a more systematic acquisition of quality requirements, the use of questionnaire-like analysis grids is promising. Through analysis grids, the acquisition step would benefit from a more structured output and would therefore facilitate the MoCQA model design step. However, the design of questionnaires that are generic enough to fit any context, yet accurate enough to produce an output that is structured and complete is not a trivial process.

[Vanderose et al., 2011] discusses how to carry out this process and introduces an iterative methodology to validate and refine the process of knowledge acquisition in the context of the MoCQA framework. This 6-step validation methodology is shown in Figure 16.1.

In Step **1**, an analysis grid is designed for each group of concepts of the quality assessment metamodel. This grid is designed according to the rules defined in

Chapter 6.
Step **2** is performed on site, with industrial partners. It is dedicated to the interviews of selected practitioners that are not familiar with the MoCQA framework. The interviews, based on the analysis grid of Step **1**, allow us to capture information about the quality assessment practices and/or goals in the company.
In Step **3**, thanks to the methods cited in Chapter 6, we align the collected information with the quality assessment metamodel concepts in order to design a MoCQA model that should reflect the quality assessment strategy of the industrial partner.
Step **4** is the cornerstone of the validation/refinement methodology and also occurs on site. This time, we interact with stakeholders *that have already been trained to use the MoCQA framework*. Those stakeholders, due to their knowledge of both the context and the MoCQA framework, can then detect the flaws or inconsistencies in our MoCQA model.
Step **5** is performed with the same stakeholders and addresses the process of checking the analysis grid itself. This validation step looks for any flaw in the appropriateness or accurateness of the questions of the grid. In this process, the practitioners that have been trained to MoCQA are indispensable since they know enough of the two worlds (i.e., industrial vs. academic) to bridge the gap and detect the errors and misconceptions we could have made.
Depending on the result of Step **5**, Step **6** consists either in designing a new version of the analysis grid leading to a new round of interviews and subsequent steps, or in refining the metamodel itself. In the latter case, the next iteration would begin with a new refined grid design.

Due to the lack of available partners to conduct this relatively heavy and time consuming process, this validation has not been carried out during the course of this research. However, providing an analysis grid that formalises the process of acquisition would benefit the effectiveness of the approach. This iterative validation methodology appears to be a good candidate to ensure that the process of designing analysis grids is consistent.

Improvement of the DSL aspect

Although UML profiles, XOCQAM and the graphical notation used by the MUG tool provide a first layer for a MoCQA-specific language, these concrete syntaxes still cannot be considered cognitively efficient. In order to improve the communicational aspects of the framework (i.e., to facilitate the exchange of information with the stakeholders), this cognitive efficiency should be refined.

The aim of this refinement process would be to provide a whole new graphical notation that is expressive enough for any stakeholder to understand the possible the exact nature of the quality profile produced during a quality assessment cycle. A research addressing this issue is currently starting as an internal collaboration

within the PReCISE research centre[3]. The aim of this collaboration is to define a graphical notation relying on pictorial representations of the measurable entity types and the possible defects that may be addressed by the quality assessment. This kind of notation should help improve the acquisition by providing a more expressive way to elicit quality-requirements, while retaining all the structural foundations of the MoCQA framework.

Process Improvement

During the course of the past decade, Agile methods have become a reliable way to improve software development. The MoCQA framework and its iterative quality assessment life-cycle support is, by design, a good candidate to support quality assessment in an Agile environment.

However, the complementarity between the two approaches could be pushed further. Although they are in essence very flexible, surveys show that Agile practitioners are in need of even more flexibility and therefore rely on the customisation of Agile methods. The problem of customising Agile methods is to provide objective ways to select the adequate methodological elements, as explained in [Ayed et al., 2012].

An Agile customisation framework integrating concepts of the MoCQA framework has already been proposed in [Ayed et al., 2012]. This metamodel has been designed to support the construction of agile methods while relying on measurement to provide guidance to agile methodologists during the method construction phase and throughout the development process. The core notion of this approach is to use measurement results from the assessment of the various deliverables produced during the process to reflect on the methodology that is currently followed. Based on this quantitative information, the Agile methodologist may be oriented towards a specific process element that is expected to correct the problems detected through measurement. In this context, the Agile customisation metamodel proposed in [Ayed et al., 2012] relies on a subset of quality assessment metamodel in order to map the measurement process with the concepts of the Open Process Framework (OPF) [OPFRO, 2009], Software Process Engineering Metamodel (SPEM) [OMG, 2008] and Standard Metamodel for Software Development Methodologies (SMSDM) [Henderson-Sellers and Gonzalez-Perez, 2005].

This research continues in an attempt to push forward the integration of the MoCQA framework with the approach proposed in [Ayed et al., 2012]. The MoCQA methodology, due to its iterative nature, could be applied as the framework in which the Agile customisation metamodel is used. Indeed, in the case of an agile method construction based on measurement values, the selection of the appropriate process element may be regarded as a hierarchy of *quality issues*. The stakeholders for these quality issues are the "agile methodologists"

[3] http://www.fundp.ac.be/en/precise/

whereas their scope would be the different part of the project where specific process elements are used. Supporting the approach this way would allow to perform a regular model-centric quality assessment. The Agile customisation metamodel would then provide the structure required to define interpretation rules that would bridge the gap between the quality indicators and the rules that address directly the methodological elements. The recommendations of the interpretation rules would consist in proposing a given process element to include in the methodology (e.g., XP programming) according to the interpretation itself (e.g., a unacceptable level of syntactic defects in the source code).

Towards automation and continuous quality assessment

Section 9.3.5 basically sets the roadmap for the tool support that has yet to be provided in order to fully support the MoCQA framework. Short-term development efforts should therefore focus on incrementally achieving this dedicated and integrated tool support. However, additional development efforts could help provide an even more complete support to the quality assurance team.

Indeed, a process the MoCQA framework that has been designed to support but has not been covered during this research is the automation of quality assessment and the continuous quality assessment of software projects during their development life-cycle. Automated and continuous model-centric quality assessment would consist in providing frequent warning and recommendations to the development team based on the central MoCQA model as the project evolves.

The measurement plan and its formalisation into a XOCQAM file provides the basis for this process. The metadata added at the operational level allows for a more systematic management of the measurement plan and the integration of various specific tools to automate some tasks (i.e., the identification of measurable entities, the calculation of the derived measures or indicators, etc.). However, many other issues have to be considered before a fully automated and continuous methodology may be implemented.

First, many measurement tools exist (Cast[4], Sonar[5], SDMetrics[6], etc.) and are already in use in given context. In order to maintain the adaptability of the framework (i.e., its ability to fit any context), the automated methodology should provide a way to bridge the gap between the external tools and the MoCQA model.

Conversely, some customised measurement or evaluation methods allowed by the MoCQA framework are not supported by any tool because they would not be useful outside a model-driven quality assessment life-cycle (i.e., too imprecise to be used as control at a later stage of the development but useful from a

[4]http://www.castsoftware.com/
[5]http://www.sonarsource.org/
[6]http://www.sdmetrics.com/

guidance perspective). In consequence, dedicated tool support for the design of customised measures should be considered in order to automate the MoCQA methodology. Existing research may be the founding step of this automation, such as the metamodel-based approach proposed in [García et al., 2007].

QuaTALOG

In essence, QuATALOG is designed to emphasise the interaction with the software quality community. Although its support is mainly based on the idea of retrieving a canonical information previously introduced, the tool is able to support the extension towards a fully community-based platform.

As a community-based platform, QuaTALOG could become a means to facilitate the exchange of information on Software Quality, both on well-established material (e.g., ISO and IEEE standards, validated metrics, etc.) and on more in-progress topics (e.g., listing of new results about a possible way to predict external quality characteristics based on internal characteristics [Bocco et al., 2005], new quality models for a specific artefacts, etc.).

From a technical point of view, this approach only requires the opening of the platform to community contributors, which is already possible in the current version of QuaTALOG. The main caveat with a community-based approach, however, is to ensure the validity of the content introduced in the knowledge base. This verification process may be implemented through various mechanisms.

Ruling out the intentional meddling of the knowledge base content (since the platform would operate with a closed community of practitioners and not a wide-open public community), the content is exposed to two types of mistakes:
First, the coexistence of more and less validated approaches within the same repository may bring confusion to the users, and bother the contributors responsible for an arguably better proposal. Fortunately, the platform already implements a mechanism to circumvent this potential issue: since the conceptual model of the repository is an adaptation of the quality assessment metamodel, it also provides a status attribute designed to characterise an assessment model, function or method. Therefore, the introduction of any of those elements requires the specification of the level of validation of the proposal (from purely experimental to empirically and theoretically validated).
Besides, the content may not be introduced accordingly to the structure defined by the quality assessment metamodel. Once again, the ontological support of the quality assessment metamodel in the design of the conceptual model of the repository provides a way to ensure that the content is introduced in a relevant manner. An additional support for the introduction could be provided by a inference engine that would provide hints based on the current content of the repository, the intent of the contributor and the quality assessment metamodel itself. This approach would push further the knowledge base aspect. Additionally,

a discussion apparatus could be implemented in order to allow community-based reflection on the relevance of given methods/function/models. This approach would push further the “wiki” aspect of the platform.

Another point of improvement for the QuaTALOG project relates to the application itself. More specifically, the web services proposed by the platform call for refinement in order to be fully satisfying. So far, the queries managed by the application are very simple and provide coarse-grained results. For instance, one may query the knowledge base in order to receive an entire quality model or a set of methods and functions linked to a specific attribute. However, the process would gain in efficiency if more complex queries were made possible (e.g., finding a candidate measurement method for a specific entity type *in order to* assess a specific quality factor, etc.).

Conclusion

As this research work started, our first observation was the enduring challenge software quality has been posing to software engineers. Despite a long and proficient record of research efforts carried out in Software Quality, defining what a "good" software is remains an elusive and difficult question that is still not solved.

This lasting effort to "corner the chimera" (an expression coined in [Dromey, 1996] and that cleverly captures the seemingly impossible task of defining quality) made the Software Quality field a very broad and dense field of study. Although it is desirable, this wealth of available methodologies and tools also induces a spread of very focused techniques that tends to isolate various quality assessment processes form each other throughout the development.

Therefore, our objective throughout this research work has been to help provide a better sense of convergence between available quality assessments. The Model-Centric Quality Assessment framework described throughout this dissertation strives to provide the quality assurance team with better methodological mechanisms to tackle the challenge of quality assessment.

Introducing the notion of model-driven quality assessment, the framework intends to support the centralisation of quality issues that may arise in a specific context. In turn, this centralisation of quality issues provides a better way for stakeholders to share a common point of view on quality in their context.

The notion of explicit and integrated quality assessment modelling supported by the framework primarily intends to take advantage of the wealth of existing quality assessment methods. Relying on a quality assessment metamodel, MoCQA models allow the integration of various specialised measurement methods and quality models into a coherent and transversal view of quality along the software development. Similarly, the explicitness of the quality assessment modelling helps extend the traditional *controlling* paradigm inherent to Software Quality. Providing each stakeholder with detailed information on the quality assessment, MoCQA models intend to involve each stakeholder (besides the quality assurance team) as a proactive actor of the quality assurance of a software project.

The notion of quality assessment life-cycle sustained by the iterative and incremental MoCQA assessment methodology strives to integrate the previous elements into a coherent whole. Iterations and increments regarding the quality assessment process allow to slowly construct a coherent set of quality requirements and monitoring methods. The methodology also emphasises the importance of a clear understanding of how measures and estimations reflect both the problems and the leads towards corrective actions.

The various case studies performed to validate the framework (and therefore its underlying principles) proved globally positive. Although it induces a methodological overhead, the framework provides enough support to streamline many activities (e.g., reusability of MoCQA models, domain-specific languages, etc.). In consequence, the framework appears as an applicable solution in an actual professional environment. The effort of systematisation provided by this research work thus provides the basis for several extensions and refinements that could steadily lead to a comprehensive and efficient quality assessment method. As the MoCQA framework intends to seemingly integrate other research efforts, this efficiency will increase as Software Quality evolves and becomes more and more successful in its search for more accurate and unambiguous measurement methods.

In the meantime, the MoCQA framework contributes to overcoming the limitations (described in Chapter 2) pertaining to Software Quality. First and foremost, the framework makes use of existing quality models in order to provide an actual *quality modelling* of software projects. Through MoCQA models, it provides a less static viewpoint on quality while not giving up the existing body of knowledge gathered in traditional quality models. The MoCQA methodology also contributes to *reducing the difficulty regarding the operationalisation of quality models*. Thanks to its operational viewpoint, it considers quality models as catalogues of quality issues that may be included more easily in a quality assessment plan.

Regarding the limitations of Software Measurement, the framework proposes to alleviate the shortcomings of measures through the documentation of their purposes and roles in the quality assessment process. The additional data collected in MoCQA models (e.g., scales, value types, etc.) provides safeguards *against the conceptual misuses of measures* by explicitly defining their purpose and how they are used in the quality assessment process. Thanks to this additional information, the quality assurance team may detect more easily a misuse or mistake regarding the choice of given measures. Similarly, MoCQA models help *alleviate the lack of validation* of measures on two levels. First, they help monitor the overall level of validation of the used measures. Additionally, they allow the evolution of the quality assessment plan towards more refined and validated measures. Finally, the framework provides a flexible mechanism that *reduces the*

difficulty to implement the underlying measurement program. First, it provides a way to communicate between stakeholders that is more standard and accepted (i.e., actual models). Moreover, the iterative methodology helps correct mistakes as the quality assessment process unfolds.

The MoCQA framework is also suited to reduce the gap between the software development and the quality assessment process. As explained in Chapter 2, the integration of quality assessment in software development raises some issues. Regarding these issues, the main advantage the framework is to provide the *definition of a proper role of quality assessment* that is more tightly integrated into software development. The introduction of a dedicated quality assessment lifecycle helps redefine the role of the quality assurance team as first-level activity. Consequently, the framework proposes to *address the organisational issues* by providing better ways to communicate within the organisation. Due to its viewpoint on software, the framework helps *bridge the gap between the viewpoint* of the development team (i.e., the model-driven or ecosystemic view on software) and the quality assurance viewpoint. The final shortcoming the framework could help solve is the *cost of quality.* Although more field-testing is required in order to actually evaluate the cost and effort of quality assessment performed with the MoCQA framework, it could provide a way to actually reduce the cost and effort of quality assessment. Thanks to the reusability of MoCQA models, efforts regarding the planning of quality assessment may be reduced. In the meantime, the focus on stakeholders' requirements could help avoid useless data collection.

In addition to its contribution to those challenges, the research work described throughout this dissertation helped us gain some more insights on Software Quality as a field. In order to bring this dissertation to an end, we would like to provide some closing comments reflecting these insights. As explained before, the fundamental question software engineers strive to answer is "how to produce reliable software?" leading to the question "what is software quality?". The design of the MoCQA framework and the different experimentations performed with this approach tend to show that defining a unique and unambiguous vision of what good software means appears to be an endless search that will never be achieved due to the fact that software engineering is in constant evolution.

However, determining if a specific software project shows a proper level of quality is not as elusive a concept. As shown in the literature and the various case studies performed with the MoCQA framework, this process is primarily influenced by contextual circumstances and heterogeneous expectations from the stakeholders. Our final conclusion regarding this research work is that the constructivist approach implemented in the framework seems to be a key aspect for the creation of a common view of the current state of a given software project, regarding its quality. Besides, allowing the iterative and incremental construction of this common view may actually help avoid unsatisfactory responses to

stakeholders' expectations.

In consequence, and according to us, a reliable software product could be defined as a software product for which all actors involved (from the management, to the developer, to the customers) share a common view of what is expected from the others, regarding quality. The MoCQA framework aims to introduce the basic set of mechanisms that are expected to support the creation of this shared view. However, it remains a small step that opens a vast field of possibilities that will require investigation in order to finally provide an answer to this existential question: what is software quality?

Bibliography

Abouzahra, A., Bézivin, J., Del Fabro, M., and Jouault, F. (2005). A practical approach to bridging domain specific languages with UML profiles. In *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA*, volume 5. Citeseer. [cited at p. 167]

Abran, A. (2010). *Software Metrics and Software Metrology*. John Wiley & Sons Interscience and IEEE-CS Press. [cited at p. 19, 26, 45, 47, 48, 51]

Abran, A. and Robillard, P. N. (1994). Function points: A study of their measurement processes and scale transformations. *Journal of Systems and Software*, 25:171–184. [cited at p. 20]

Abran, A. and Sellami, A. (2002). Measurement and metrology requirements for empirical studies in software engineering. In *STEP '02: Proceedings of the 10th International Workshop on Software Technology and Engineering Practice*, page 185, Washington, DC, USA. IEEE Computer Society. [cited at p. 19, 39]

Al Balushi, T. H., Sampaio, P. R. F., Dabhi, D., and Loucopoulos, P. (2007). ElicitO: a quality ontology-guided NFR elicitation tool. In *Proceedings of the 13th international working conference on Requirements engineering: foundation for software quality*, REFSQ'07, pages 306–319, Berlin, Heidelberg. Springer-Verlag. [cited at p. 137]

Albrecht, A. (1979). Measuring application development productivity. In Press, I. B. M., editor, *IBM Application Development Symp.*, pages 83–92. [cited at p. 20]

Antoniol, G., Lokan, C., Caldiera, G., and Fiutem, R. (1999). A function point-like measure for object-oriented software. *Empirical Software Engineering*, 4(3):263–287. [cited at p. 21]

Antony, J. (2004). Some pros and cons of six sigma: an academic perspective. *The TQM Magazine*, 16(4):303–306. [cited at p. 28, 29]

Ayed, H., Vanderose, B., and HABRA, N. (2012). A metamodel-based approach for customizing and assessing agile methods. In *Proceedings of the 8th International Conference on the Quality of Information and Communications Technology (QUATIC 2012)*. [cited at p. 267]

Bansiya, J. and Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17. [cited at p. 21]

Bansiya, J., Etzkorn, L. H., Davis, C. G., and Li, W. (1999). A class cohesion metric for object-oriented designs. *JOOP*, 11(8):47–52. [cited at p. 21]

Basili, V., Caldiera, G., and Rombach, D. H. (1994). The goal question metric approach. [cited at p. 24, 31, 73]

Basili, V. R. and Weiss, D. M. (1984). A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, 10(6):728–738. [cited at p. 18, 66]

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for Agile Software Development. [cited at p. 64]

Behkamal, B., Kahani, M., and Akbari, M. K. (2009). Customizing ISO 9126 quality model for evaluation of B2B applications. *Information and Software Technology*, 51(3):599 – 609. [cited at p. 14]

Bengtsson, P., Bengtsson, P., and Bengtsson, P. (2002). Architecture-level modifiability analysis. *Journal of Systems and Software*, 69. [cited at p. 136, 212]

Biehl, R. E. (2004). Six sigma for software. *IEEE Softw.*, 21(2):68–70. [cited at p. 29]

Biggerstaff, T. J. and Richter, C. (1989). Software reusability: vol. 1, concepts and models. chapter Reusability framework, assessment, and directions, pages 1–17. ACM, New York, NY, USA. [cited at p. 68]

Bocco, M. G., Moody, D. L., and Piattini, M. (2005). Assessing the capability of internal metrics as early indicators of maintenance effort through experimentation: Research articles. *J. Softw. Maint. Evol.*, 17(3):225–246. [cited at p. 155, 269]

Bøegh, J., Depanfilis, S., Kitchenham, B., and Pasquini, A. (1999). A method for software quality planning, control, and evaluation. *IEEE Softw.*, 16(2):69–77. [cited at p. 25]

Boehm, B. W. (1981). *Software Engineering Economics.* Prentice Hall PTR, Upper Saddle River, NJ, USA. [cited at p. 11]

Boloix, G., Sorenson, P. G., and Tremblay, J. P. (1993). Software metrics using a metasystem approach to software specification. *J. Syst. Softw.*, 20(3):273–294. [cited at p. 20]

Booch, G. (2004). *Object-Oriented Analysis and Design with Applications (3rd Edition).* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA. [cited at p. 65]

Briand, L., Devanbu, P., and Melo, W. (1997a). An investigation into coupling measures for C++. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 412–421, New York, NY, USA. ACM. [cited at p. 21]

Briand, L. and Morasca, S. (1997). Software measurement and formal methods: A case study centered on TRIO+ specifications. In *Proc. First Intĺ Conf. Formal Eng. Methods (ICFEM 9́7*, pages 12–14. [cited at p. 20]

Briand, L. C., Differding, C. M., and Rombach, H. D. (1997b). Practical guidelines for measurement-based process improvement. *Special issue of International Journal of Software Engineering & Knowledge Engineering.* [cited at p. 48]

Briand, L. C., Morasca, S., and Basili, V. R. (1996). Property-based software engineering measurement. *IEEE Trans. Softw. Eng.*, 22(1):68–86. [cited at p. 19, 21]

Briand, L. C., Morasca, S., and Basili, V. R. (1999). Defining and validating measures for object-based high-level design. *IEEE Trans. Software Eng.*, 25(5):722–743. [cited at p. 21]

Briand, L. C., Morasca, S., and Basili, V. R. (2002). An operational process for goal-driven definition of measures. *IEEE Trans. Softw. Eng.*, 28(12):1106–1125. [cited at p. 24, 52, 54, 65]

Brito e Abreu, F. and Carapuça, R. (1994a). Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1):87–96. [cited at p. 21]

Brito e Abreu, F. and Carapuça, R. (1994b). Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of the 4th Interntional Conference on Software Quality*, USA. McLean. [cited at p. 21]

Brito e Abreu, F. and Melo, W. (1996). Evaluating the impact of object-oriented design on software quality. *Software Metrics, IEEE International Symposium on*, 0:90. [cited at p. 21]

Budgen, D. (2003). *Software Design.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. [cited at p. 210]

Bundschuh, M. and Dekkers, C. (2008). *The IT Measurement Compendium: Estimating and Benchmarking Success with Functional Size Measurement*, chapter Variants of the IFPUG Function Point Counting Method, pages 397–407. Springer Publishing Company, Incorporated. [cited at p. 20]

Cachero, C., Calero, C., and Poels, G. (2007). *Metamodeling the Quality of the Web Development Process Intermediate Artifacts*, pages 74–89. [cited at p. 31]

Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493. [cited at p. 21]

Chirinos, L., Losavio, F., and Bøegh, J. (2005). Characterizing a data model for software measurement. *Journal of Systems and Software*, 74(2):207 – 226. The new context for software engineering education and training. [cited at p. 17, 25, 52, 54]

Christel, M. and Kang, K. (1992). Issues in requirements elicitation. Technical report. [cited at p. 137]

Clements, P., Kazman, R., and Klein, M. (2001). *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley. [cited at p. 136]

Cockburn, A. (2006). *Agile Software Development: The Cooperative Game (2nd Edition) (Agile Software Development Series)*. Addison-Wesley Professional. [cited at p. 64]

Conradi, R. and Fuggetta, A. (2002). Improving software process improvement. *IEEE Softw.*, 19(4):92–99. [cited at p. 43]

Cook, S. (1996). *Process Improvement: A Handbook for Managers*. Gower. [cited at p. 27]

Cruz-Lemus, J. A., Genero, M., Manso, M. E., Morasca, S., and Piattini, M. (2009). Assessing the understandability of UML statechart diagrams with composite states - A family of empirical studies. *Empirical Software Engineering*, 14(6):685–719. [cited at p. 22]

Cyra, L. and Górski, J. (2008). Extending GQM by argument structures. pages 26–39. [cited at p. 24]

Deissenböck, F. (2009). *Continuous Quality Control of Long-Lived Software Systems*. PhD thesis, Institut für Informatik der Technischen Universität München. [cited at p. 32, 49, 81, 84]

Deissenboeck, F., Juergens, E., Lochmann, K., and Wagner, S. (2009). Software quality models: purposes, usage scenarios and requirements. In *Proceedings of the Seventh ICSE conference on Software quality*, WOSQ'09, pages 9–14, Washington, DC, USA. IEEE Computer Society. [cited at p. 35, 37]

Deming, W. E. (2000). *Out of the Crisis*. MIT Press. Paperback. Originally published by MIT-CAES in 1982. [cited at p. 28]

Derr, K. W. (1995). *Applying OMT: a practical step-by-step guide to using the object modeling technique*. SIGS Publications, Inc., New York, NY, USA. [cited at p. 22]

Dromey, R. G. (1995). A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2):146–162. [cited at p. 12]

Dromey, R. G. (1996). Cornering the chimera. *IEEE Softw.*, 13(1):33–43. [cited at p. 12, 36, 65, 76, 124, 271]

Dubielewicz, I., Hnatkowska, B., Huzar, Z., and Tuzinkiewicz, L. (2006). Software quality metamodel for requirement, evaluation and assessment. In *ISIM06 Conference*, volume No. 105, pages 115–122, Prerov, Czech Republic, Acta Mosis. [cited at p. 33]

Englebert, V. and Heymans, P. (2007). Towards more extensible metaCASE tools. In *Advanced Information Systems Engineering*, pages 454–468. Springer. [cited at p. 176]

Falcone, G. (2010). *Hierarchy-aware software metrics in component composition hierarchies*. PhD thesis, Berlin. [cited at p. 156]

Fenton, N., Neil, M., Marsh, W., Hearty, P., Marquez, D., Krause, P., and Mishra, R. (2007). Predicting software defects in varying development lifecycles using Bayesian nets. *Inf. Softw. Technol.*, 49(1):32–43. [cited at p. 16]

Fenton, N. E. and Neil, M. (1999). Software metrics: success, failures and new directions. *J. Syst. Softw.*, 47(2-3):149–157. [cited at p. 1, 42]

Fenton, N. E. and Pfleeger, S. L. (1998). *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA. [cited at p. 17, 18, 39, 46, 49]

Finney, K., Rennolls, K., and Fedorec, A. (1998). Measuring the comprehensibility of Z specifications. *Journal of Systems and Software*, 42(1):3 – 15. [cited at p. 20]

Firesmith, D. (2005). Are your requirements complete? *Journal of Object Technology*, 4(1):27–44. [cited at p. 225]

García, F., Bertoa, M. F., Calero, C., Vallecillo, A., Ruíz, F., Piattini, M., and Genero, M. (2006). Towards a consistent terminology for software measurement. *Information and Software Technology*, 48(8):631 – 644. [cited at p. 19, 26, 31, 45, 46, 47, 48, 52, 53]

García, F., Serrano, M., Cruz-Lemus, J., Ruíz, F., and Piattini, M. (2007). Managing software process measurement: A metamodel-based approach. *Information Sciences*, 177(12):2570–2586. [cited at p. 31, 269]

García Frey, A., Céret, E., Dupuy-Chessa, S., and Calvary, G. (2011). QUIMERA: a quality metamodel to improve design rationale. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '11, pages 265–270, New York, NY, USA. ACM. [cited at p. 33]

Genero, M. (2002). *Defining and Validating Metrics for Conceptual Models*. PhD thesis, University of Castilla-La Mancha. [cited at p. 22]

Genero, M., Manso, E., Visaggio, A., Canfora, G., and Piattini, M. (2007). Building measure-based prediction models for UML class diagram maintainability. *Empirical Softw. Engg.*, 12(5):517–549. [cited at p. 22]

Genero, M., Piattini, M., and Calero, C. (2000). Early measures for UML class diagrams. *L'OBJET*, 6(4). [cited at p. 22]

Genero, M., Piattini, M., and Calero, C., editors (2005a). *Metrics For Software Conceptual Models*. World Scientific Publishing Co., Inc., River Edge, NJ, USA. [cited at p. 20, 22, 23]

Genero, M., Piattini, M., and Calero, C. (2005b). A survey of metrics for UML class diagrams. *Journal of Object Technology*, 4:59–92. [cited at p. 21, 22]

Gerhardt-Powals, J. (1996). Cognitive engineering principles for enhancing human-computer performance. *Int. J. Hum.-Comput. Interact.*, 8(2):189–211. [cited at p. 171]

Gilson, F. and Englebert, V. (2011). Rationale, decisions and alternatives traceability for architecture design. In *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*, ECSA '11, pages 4:1–4:9, New York, NY, USA. ACM. [cited at p. 210]

Grady, R. B. (1992). *Practical software metrics for project management and process improvement*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. [cited at p. 12]

Grady, R. B. and Caswell, D. L. (1987). *Software metrics: establishing a company-wide program*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. [cited at p. 12]

Habra, N., Abran, A., Lopez, M., and Sellami, A. (2008). A framework for the design and verification of software measurement methods. *J. Syst. Softw.*, 81(5):633–648. [cited at p. 19, 38, 39, 45, 46, 47, 49, 50]

Habra, N. and Lopez, M. (2004). A structured analysis of the McCabe cyclomatic complexity measure. In Dumbke, R. and Abran, A., editors, *14th International Workshop on Software Measurement (IWSM2004)*, pages –, Aachen. Shaker Verlag. [cited at p. 38]

Halstead, M. H. (1977). *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA. [cited at p. 23]

Hanoteau, S. (2012). Déploiement de l´approche MoCQA en environnement professionnel. Master's thesis, University of Namur. [cited at p. 241, 246, 248]

Harrison, R., Counsell, S., and Nithi, R. (1998). Coupling metrics for object-oriented design. In *METRICS '98: Proceedings of the 5th International Symposium on Software Metrics*, page 150, Washington, DC, USA. IEEE Computer Society. [cited at p. 21]

Henderson-Sellers, B. and Gonzalez-Perez, C. (2005). A comparison of four process metamodels and the creation of a new generic standard. *Information and software technology*, 47(1):49–65. [cited at p. 267]

Henry, S. and Kafura, D. (1981). Software structure metrics based on information flow. *IEEE Trans. Softw. Eng.*, 7(5):510–518. [cited at p. 22]

IEEE (1998). Standard for a software quality metrics methodology. *IEEE Computer Society, IEEE Std*, pages 1061–1998. [cited at p. 46, 48, 49]

IEEE Computer Society (2004). *Software Engineering Body of Knowledge (SWEBOK)*. Angela Burgess, EUA. [cited at p. 94]

ISO/IEC (1999). 14598-1:1999 Information technology – Software product evaluation – Part 1: General overview. [cited at p. 49, 89, 193]

ISO/IEC (2001a). 9126-1, Software engineering - product quality - Part 1: Quality Model. [cited at p. 13, 47, 48, 49, 219]

ISO/IEC (2001b). 9126-2, Software engineering - product quality - Part 2: External metrics. [cited at p. 23, 191]

ISO/IEC (2001c). 9126-3, Software engineering - product quality - Part 3: Internal Metrics. [cited at p. 23, 191]

ISO/IEC (2001d). 9126-4, Software engineering - product quality - Part 4: Quality In Use Metrics. [cited at p. 23, 191]

ISO/IEC (2003a). 15504-2:2003 Information technology - Process assessment - Part 2: Performing an assessment. [cited at p. 30]

ISO/IEC (2003b). 19761:2003, Software engineering – COSMIC-FFP – A functional size measurement method. [cited at p. 20]

ISO/IEC (2005a). 19502:2005 Information technology - Meta Object Facility (MOF). [cited at p. 82]

ISO/IEC (2005b). 25000, Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE. [cited at p. 14]

ISO/IEC (2007a). 15939:2007 Systems and software engineering - Measurement process. [cited at p. 19, 52, 86, 185]

ISO/IEC (2007b). Guide 99:2007, International vocabulary of metrology – Basic and general concepts and associated terms (VIM). [cited at p. 19, 47, 48, 50]

ISO/IEC (2008). 12207:2008 – Systems and software engineering – Software life cycle processes. [cited at p. 40, 50, 51]

ISO/IEC (2011). 25010, Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. [cited at p. 222]

Jacquet, J.-P. and Abran, A. (1997). From software metrics to software measurement methods: A process model. In *Proceedings of the 3rd International Software Engineering Standards Symposium (ISESS '97)*, page 128, Washington, DC, USA. IEEE Computer Society. [cited at p. 18]

Jonassen, D. (1991). Objectivism versus constructivism: Do we need a new philosophical paradigm? *Educational Technology Research and Development*, 39:5–14. 10.1007/BF02296434. [cited at p. 63]

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., and Valduriez, P. (2006). ATL: A QVT-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 719–720. ACM. [cited at p. 168]

Juristo, N. and Moreno, A. M. (2001). *Basics of Software Engineering Experimentation.* Kluwer Academic Publishers, Boston, USA. [cited at p. 183]

Kaner, C., Member, S., and Bond, W. P. (2004). Software engineering metrics: What do they measure and how do we know? In *In METRICS 2004. IEEE CS.* Press. [cited at p. 125]

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-Oriented Domain Analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute. [cited at p. 94]

Kantner, R. (2000). *The ISO 9000: Answer Book, 2nd Edition.* John Wiley & Sons, Inc. [cited at p. 29]

Kasunic, M. (2006). The state of software measurement practice: Results of 2006 survey. Technical report, Software Engineering Institute. [cited at p. 1, 40, 42]

Kazman, R., Kazman, R., Klein, M., Klein, M., Clements, P., Clements, P., Compton, N. L., and Col, L. (2000). ATAM: Method for architecture evaluation. [cited at p. 199, 200, 201, 202]

Kelly, S. (2004). Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM. In *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Workshop on Best Practices for Model Driven Software Development.* [cited at p. 175]

Kent, S. (2002). Model Driven Engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK. Springer-Verlag. [cited at p. 165]

Khosravi, K. and Guéhéneuc, Y.-G. (2005). Open issues with quality models. In Brito e Abreu, F., Calero, C., Lanza, M., Poels, G., and Sahraoui, H. A., editors, *Proceedings of the 9th ECOOP workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE).* Springer-Verlag. [cited at p. 33]

Kitchenham, B., Pfleeger, S. L., and Fenton, N. (1995). Towards a framework for software measurement validation. *IEEE Trans. Softw. Eng.*, 21(12):929–944. [cited at p. 18, 21, 39]

Klaes, M., Lampasona, C., and Nunnenmacher, S. (2010). How to evaluate meta-models for software quality? In *Proceedings of the 20th International Workshop on Software Measurement (IWSM2010).* [cited at p. 33, 262]

Koshima, A., Englebert, V., and Thiran, P. (2011). Distributed collaborative model editing framework for Domain Specific Modeling tools. In *Global Software Engineering (ICGSE), 2011 6th IEEE International Conference on*, pages 113–118. IEEE. [cited at p. 168]

Koziolek, H. (2011). Sustainability evaluation of software architectures: a systematic review. In *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*, QoSA-ISARCS '11, pages 3–12, New York, NY, USA. ACM. [cited at p. 39, 136, 199]

Krogstie, J., Lindland, O. I., and Sindre, G. (1995). Defining quality aspects for conceptual models. In *Proceedings of the IFIP international working conference on Information system concepts*, pages 216–231, London, UK, UK. Chapman & Hall, Ltd. [cited at p. 17]

Kurtev, I. (2008). State of the art of QVT: A model transformation language standard. *Applications of Graph Transformations with Industrial Relevance*, pages 377–393. [cited at p. 168]

Lange, C. and Chaudron, M. (2004). An empirical assessment of completeness in UML designs. *IEE Seminar Digests*, 2004(920):111–119. [cited at p. 22]

Lange, C. F. J. (2007). *Assessing and Improving the Quality of Modeling.* PhD thesis, Technische Universiteit Eindhoven. [cited at p. 22]

Lavazza, L. and Barresi, G. (2005). Automated support for process-aware definition and execution of measurement plans. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 234–243, New York, NY, USA. ACM. [cited at p. 24]

Lee, Y. and Chang, K. H. (2000). Reusability and maintainability metrics for object-oriented software. In *ACM-SE 38: Proceedings of the 38th annual on Southeast regional conference*, pages 88–94, New York, NY, USA. ACM. [cited at p. 33]

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and Laws of Software Evolution - The Nineties View. *Software Metrics, IEEE International Symposium on*, 0:20+. [cited at p. 152]

Li, E. Y., Chen, H. G., and Cheung, W. (2000). Total Quality Management in Software Development Process. *The Journal of the Quality Assurance Institute*, 14(1). [cited at p. 27, 28]

Li, W. and Henry, S. (1993). Object-oriented metrics that predict maintainability. *J. Syst. Softw.*, 23(2):111–122. [cited at p. 21]

Lindland, O. I., Sindre, G., and Solvberg, A. (1994). Understanding quality in conceptual modeling. *IEEE Softw.*, 11(2):42–49. [cited at p. 16]

Lopez, M., Paulus, V., and Habra, N. (2003). Integrated validation process of software measure. In *Proceedings of the 13th International Workshop on Software Measurement (IWSM2003)*, MontrÃl'al, Canada. Shaker Verlag. [cited at p. 18]

Lorenz, M. and Kidd, J. (1994). *Object-oriented software metrics: a practical guide.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA. [cited at p. 21]

Losavio, F., Chirinos, L., Matteo, A., Lévy, N., and Ramdane-Cherif, A. (2004). ISO quality standards for measuring architectures. *Journal of Systems and Software*, 72(2):209–223. [cited at p. 14]

Losavio, F., Chirinos, L., and Perez, M. A. (2001). Quality models to design software architectures. In *Proc. Technology of Object-Oriented Languages and Systems TOOLS 38*, pages 123–135. [cited at p. 14]

Lungu, M. (2009). *Reverse Engineering Software Ecosystems.* PhD thesis, University of Lugano, Switzerland. [cited at p. 2, 41]

Marchesi, M. (1998). OOA metrics for the Unified Modeling Language. In *CSMR '98: Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*, page 67, Washington, DC, USA. IEEE Computer Society. [cited at p. 20, 21]

Matulevicius, R., Kamseu, F., and Habra, N. (2009). Measuring open source documentation availability. In *Proceedings of the international Conference on Quality Engineering in Software Technology.* [cited at p. 23, 193, 197]

McCabe, T. J. (1976). A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA. IEEE Computer Society Press. [cited at p. 23]

Mccall, J. A., Richards, P. K., and Walters, G. F. (1977). Factors in software quality. Volume i. concepts and definitions of software quality. Technical Report ADA049014, General Electric co Sunnyval, Ca. [cited at p. 11, 49, 219]

Mens, T., Czarnecki, K., and Gorp, P. V. (2005a). 04101 discussion – a taxonomy of model transformations. In Bezivin, J. and Heckel, R., editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. [cited at p. 99]

Mens, T., Doctors, L., Habra, N., Vanderose, B., and Kamseu, F. (2011). Qualgen: Modeling and analysing the quality of evolving software systems. In *Proc. IEEE Int'l CSMR*, pages 351 – 354. IEEE. [cited at p. 132]

Mens, T. and Goeminne, M. (2011). Analysing the evolution of social aspects of open source software ecosystems. In *Proc. 3rd Int. Workshop on Software Ecosystems (IWSECO)*, pages 1–14. [cited at p. 2]

Mens, T. and Gorp, P. V. (2006). A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125 – 142. Proceedings of the International Workshop on Graph and Model Transformation (GraMoT 2005). [cited at p. 141, 174, 224]

Mens, T. and Lanza, M. (2002). A graph-based metamodel for object-oriented software metrics. *Electr. Notes Theor. Comput. Sci.*, 72(2). [cited at p. 33]

Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., and Jazayeri, M. (2005b). Challenges in software evolution. In *IWPSE '05: Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 13–22, Washington, DC, USA. IEEE Computer Society. [cited at p. 152, 168]

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344. [cited at p. 69]

Meyer, M. H. and Lehnerd, A. P. (1997). *The Power of Product Platforms.* Free Press, New York. [cited at p. 235]

Mich, L., Anesi, C., and Berry, D. M. (2004). Requirements engineering and creativity: An innovative approach based on a model of the pragmatics of communication. In *in Proceedings of Requirements Engineering: Foundation of Software Quality REFSQâĂŹ04.* [cited at p. 137]

Miller, J. and Mukerji, J. (2003). MDA Guide version 1.0.1. Technical report, Object Management Group (OMG). [cited at p. 83, 98]

Miller, R. E. (2009). *The Quest for Software Requirements.* MavenMark Books, USA. [cited at p. 137]

Miranda, D., Genero, M., and Piattini, M. (2003). Empirical validation of metrics for UML statechart diagrams. In *ICEIS (1)*, pages 87–95. [cited at p. 22]

Mohagheghi, P. and Dehlen, V. (2008). Developing a quality framework for model-driven engineering. [cited at p. 2, 15, 33]

Mohagheghi, P., Dehlen, V., and Neple, T. (2008). A metamodel and supporting process and tool for specifying quality models in model-based software development. *Nordic J. of Computing*, 14(4):301–320. [cited at p. 33]

Moody, D. L. (2009). The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 35:756–779. [cited at p. 171]

Moody, D. L. and Shanks, G. G. (2003). Improving the quality of data models: empirical validation of a quality management framework. *Information Systems*, 28(6):619–650. [cited at p. 15]

Mora, B., Piattini, M., Ruiz, F., and Garcia, F. (2008). SMML: Software Measurement Modeling Language. In *Proceedings of the 8th Workshop on Domain-Specific Modeling (DSM'2008)*. [cited at p. 31, 69]

Morasca, S. (1999). Measuring attributes of concurrent software specifications in Petri nets. *Software Metrics, IEEE International Symposium on*, 0:100. [cited at p. 20]

Morasca, S. (2001). *Handbook of Software Engineering And Knowledge Engineering: Recent Advances*, chapter 2: Software Measurement, pages 239–276. World Scientific Publishing Co., Inc., River Edge, NJ, USA. [cited at p. 18, 20, 23]

Morasca, S. (2008). Refining the axiomatic definition of internal software attributes. In *ESEM '08: Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 188–197, New York, NY, USA. ACM. [cited at p. 19]

Morasca, S. and Briand, L. (1997). Towards a theoretical framework for measuring software attributes. In *Proc. IEEE Symp. Software Metrics*, pages 119–126. IEEE. [cited at p. 19, 39]

Neil, M. and Fenton, N. (1996). Predicting software quality using Bayesian belief networks. In *Proc 21st Ann. Software Eng. Workshop, NASA Goddard Space Flight Centre*, pages 217–230. [cited at p. 16]

Neil, M., Fenton, N., and Nielson, L. (2000). Building large-scale Bayesian networks. *Knowl. Eng. Rev.*, 15(3):257–284. [cited at p. 16]

OMG (2006). *Meta Object Facility (MOF) Core Specification Version 2.0*. [cited at p. 83]

OMG (2008). Software & Systems Process Engineering Metamodel Specification (SPEM) version 2. [cited at p. 267]

OMG (2010). OCL 2.2 Specification. [cited at p. 167]

OPFRO (2009). Open process framework. [cited at p. 267]

Ortega, M., Pérez, M., and Rojas, T. (2003). Construction of a systemic quality model for evaluating a software product. *Software Quality Control*, 11(3):219–242. [cited at p. 11, 12, 14, 192, 221]

Parnas, D. L. (1994). Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA. IEEE Computer Society Press. [cited at p. 152]

Paulk, M. C. (1999). Analyzing the conceptual relationship between ISO/IEC 15504 (Software Process Assessment) and the Capability Maturity Model for Software. In *in Proceedings, Ninth International Conference on Software Quality*, pages 4–6. [cited at p. 30]

Perez Garcia, F., Pinna Puissant, J., Mens, T., Kamseu, F., and Habra, N. (2012). Software quality practices in industry: A pilot study in Wallonia. Technical report. [cited at p. 37]

Peters, J. F. and Pedrycz, W. (1998). *Software Engineering: An Engineering Approach*. John Wiley & Sons, Inc., New York, NY, USA. [cited at p. 1]

Pfleeger, S. L. (1998). *Software Engineering: Theory and Practice*. Prentice Hall. [cited at p. 11]

PMI (2004). *A Guide To The Project Management Body Of Knowledge (PMBOK Guides)*. Project Management Institute. [cited at p. 248]

Pohl, K., Böckle, G., and Linden, F. J. v. d. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. [cited at p. 236]

Prather, R. E. (1984). An axiomatic theory of software complexity measure. *The Computer Journal*, 27(4):340–347. [cited at p. 19]

Pressman, R. S. (2000). *Software Engineering : A Practioner's Approach*. Mc Graw-Hill International (UK) Limited. [cited at p. 37]

Ramdoyal, R., Cleve, A., and Hainaut, J.-L. (2010). Reverse engineering user interfaces for interactive database conceptual analysis. In *Proceedings of the 22nd international conference on Advanced information systems engineering*, CAiSE'10, pages 332–347, Berlin, Heidelberg. Springer-Verlag. [cited at p. 135]

Read, D. (2005). Iterative development: Key technique for managing software developments. In *Proceedings of ICT WA '05*. [cited at p. 64]

Reynoso, L., Cruz-Lemus, J. A., Genero, M., and Piattini, M. (2008). Formal definition of measures for UML statechart diagrams using OCL. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 846–847, Fortaleza, Ceara, Brazil. ACM. [cited at p. 22]

Riguzzi, F. (1996). A survey of software metrics. Technical Report DEIS-LIA-96-010, LIA Series n.17, DEIS, Università di Bologna. [cited at p. 2]

Rittgen, P. (2008). COMA: A tool for collaborative modeling. In *CAiSE Forum*, pages 61–64. [cited at p. 168]

Rittgen, P. (2009). Collaborative modeling - A design science approach. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–10. IEEE. [cited at p. 168]

Robles, G., Gonzalez-Barahona, J. M., and Merelo, J. J. (2006). Beyond source code: the importance of other artifacts in software development (a case study). *J. Syst. Softw.*, 79(9):1233–1248. [cited at p. 41]

Rooney, J. J., Heuvel, V., and Lee, N. (2004). Root Cause Analysis for beginners. *Quality Progress*, 37(7):45–53. [cited at p. 151]

RTCA (1992). DO-178b, Software Considerations in Airborne Systems and Equipment Certification. *Radio Technical Commission for Aeronautics (RTCA), European Organization for Civil Aviation Electronics (EUROCAE), DO178-B.* [cited at p. 235]

Saeki, M. (2003). Embedding metrics into information systems development methods: An application of method engineering technique. In *CAiSE*, pages 374–389. [cited at p. 20]

Schmidt, D. C. (2006). Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):25–31. [cited at p. 2, 37]

SEI (2010). CMMI for Development, Version 1.3,. Technical report, CMU/SEI-2010-TR-033, Carnegie Mellon University. [cited at p. 47, 50, 51, 63]

Sendall, S. and Kozaczynski, W. (2003). Model transformation: The heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45. [cited at p. 168]

Sharp, H., Finkelstein, A., and Galal, G. (1999). Stakeholder identification in the requirements engineering process. In *Proceedings of the 10th International Workshop on Database & Expert Systems Applications*, DEXA '99, pages 387–, Washington, DC, USA. IEEE Computer Society. [cited at p. 65, 130]

Si-Said Cherfi, S., Akoka, J., and Comyn-Wattiau, I. (2007). Perceived vs. measured quality of conceptual schemas: an experimental comparison. In *ER '07: Tutorials, posters, panels and industrial contributions at the 26th international conference on Conceptual modeling*, pages 185–190, Darlinghurst, Australia, Australia. Australian Computer Society, Inc. [cited at p. 22]

Sprinkle, J. M., Ledeczi, A., Karsai, G., and Nordstrom, G. (2001). The new metamodeling generation. *Engineering of Computer-Based Systems, IEEE International Conference on the*, 0:0275. [cited at p. 82]

Staples, M., Niazi, M., Jeffery, R., Abrahams, A., Byatt, P., and Murphy, R. (2007). An exploratory study of why organizations do not adopt CMMI. *J. Syst. Softw.*, 80(6):883–895. [cited at p. 31]

Stelzer, D., Mellis, W., and Herzwurm, G. (1996). Software process improvement via ISO 9000? Results of two surveys among European Software Houses. In *Proceedings of the 29th Hawaii International Conference on System Sciences Volume 1: Software Technology and Architecture*, HICSS '96, pages 703–, Washington, DC, USA. IEEE Computer Society. [cited at p. 29]

Stevens, S. S. (1975). Psychophysics: Introduction to its perceptual, neural, and social prospects. [cited at p. 17]

Suryn, W., Abran, A., and April, A. (2003). ISO/IEC SQuaRE: The second generation of standards for software product quality. In *Proceedings of the 7th IASTED International Conference on Software Engineering and Applications (ICSEAâĂŹ03*, pages 1–9. [cited at p. 14]

Symons, C. R. (1991). *Software sizing and estimating: Mk II FPA (Function Point Analysis)*. John Wiley & Sons, Inc., New York, NY, USA. [cited at p. 20]

Tian, J. and Zelkowitz, M. V. (1992). A formal program complexity model and its application. *Journal of Systems and Software*, 17(3):253 – 266. [cited at p. 19]

Torchiano, M., Jaccheri, L., Sørensen, C.-F., and Wang, A. I. (2002). COTS products characterization. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 335–338, New York, NY, USA. ACM. [cited at p. 14]

Torgerson, W. S. (1958). Theory and methods of scaling. New York: John Wiley & Sons, 1958. [cited at p. 17]

Trendowicz, A. and Punter, T. (2003). Quality modeling for software product lines. In *In: 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE03)*. [cited at p. 15]

Uschold, M. and King, M. (1995). Towards a methodology for building ontologies. In *In Workshop on Basic Ontological Issues in Knowledge Sharing, held in conjunction with IJCAI-95*. [cited at p. 51, 52]

van Amstel, M., Lange, C., and van den Brand, M. (2009). Using metrics for assessing the quality of ASF+SDF model transformations. *Theory and Practice of Model Transformations*, pages 239–248. [cited at p. 23]

van Lamsweerde, A. and Letier, E. (2004). From object orientation to goal orientation: A paradigm shift for requirements engineering. *Radical Innovations of Software and Systems Engineering in the Future*, pages 153–166. [cited at p. 137]

van Opzeeland, D. J., Lange, C. F., and Chaudron, M. R. (2005). Quantitative techniques for the assessment of correspondence between UML designs and implementations. In *Proc. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 1–17. [cited at p. 22]

Vanderose, B. and Habra, N. (2008). Towards a generic framework for empirical studies of model-driven engineering. In *Proceedings of the First Workshop on Empirical Studies of Model-Driven Engineering Toulouse, France, September 29, 2008.* [cited at p. 90]

Vanderose, B. and Habra, N. (2011). Tool-support for a model-centric quality assessment: Quatalog. In *Proceedings of the 2011 Joint Conference of the 21st International Workshop on Software Measurement and the 6th International Conference on Software Process and Product Measurement*, IWSM-MENSURA '11, pages 263–268, Washington, DC, USA. IEEE Computer Society. [cited at p. 173]

Vanderose, B., HABRA, N., and Kamseu, F. (2011). Operationalization of a model-centric quality assessment (MoCQA) framework. In *Proceedings of the 3rd Workshop on Leveraging Empirical Research Results for Software Business Success (EPIC2011)*. [cited at p. 265]

Vanderose, B., Kamseu, F., and Habra, N. (2010). Towards a model-centric quality assessment. In *Proceedings of the 20th International Workshop on Software Measurement (IWSM2010)*, pages 21–34. [cited at p. 71, 175, 215]

Vanderose, B., Mens, T., Kamseu, F., and HABRA, N. (2012). A feasability study of quality assessment during software maintenance. In *Proceedings of the 6th International Workshop on Software Quality and Maintainability (SQM2012)*. [cited at p. 219]

Vaucher, S. (2010). *Modelling Software Quality: A Multidimensional Approach*. PhD thesis, Université de Montréal. [cited at p. 262]

Vignaga, A. (2007). A methodological approach to developing model transformations. In *MoDELS (Doctoral Symposium)*. [cited at p. 2]

Wagner, S., Lochmann, K., Heinemann, L., Kläs, M., Trendowicz, A., Plösch, R., Seidl, A., Goeb, A., and Streit, J. (2012). The Quamoco product quality modelling and assessment approach. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1133–1142, Piscataway, NJ, USA. IEEE Press. [cited at p. 32]

Wagner, S., Lochmann, K., Winter, S., Goeb, A., and Klaes, M. (2009). Quality models in practice: A preliminary analysis. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 464–467, Washington, DC, USA. IEEE Computer Society. [cited at p. 36, 133]

Westfall, L. and Road, C. (2005). 12 steps to useful software metrics. *Proceedings of the Seventeenth Annual Pacific Northwest Software Quality Conference*, 57 Suppl 1(May 2006):S40–3. [cited at p. 2, 42, 51, 65, 66, 69, 70, 130, 252]

Weyuker, E. J. (1988). Evaluating software complexity measures. *IEEE Trans. Softw. Eng.*, 14(9):1357–1365. [cited at p. 19]

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2000). *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, Norwell, MA, USA. [cited at p. 183, 219]

Yu, E. and Mylopoulos, J. (1998). Why goal-oriented requirements engineering. In *Proceedings of the 4th International Workshop on Requirements Engineering: Foundations of Software Quality*, pages 15–22. [cited at p. 137]

Zeiss, B., Vega, D., Schieferdecker, I., Neukirchen, H., and Grabowski, J. (2007). Applying the ISO 9126 quality model to test specifications exemplified for TTCN-3 test specifications. In *Software Engineering*, pages 231–242. [cited at p. 14]

Zuse, H. (1997). *A Framework of Software Measurement*. Walter de Gruyter & Co. [cited at p. 18, 39, 156]

Index

A

analysis model, 48
artefact, 90
artefact type, 90
assessment model, 115
attribute, 46

B

base measure, 47
behaviour, 94
behaviour type, 94

D

decision criteria, 48
deliverable, 50
derivation, 98
derivation type, 97
derived measure, 47

E

entity, 45
entity class, 46
entity population, 46
external attribute, 46

I

indicator, 48
information need, 48
internal attribute, 46
interpretation rule, 120

M

measure, 47
measurement function, 47
measurement life cycle, 47, 62
measurement method, 48
measurement plan, 48
measurement procedure, 48
metric, 48
MoCQA model, 72, 76, 81

O

operational customised quality assessment model, 61, 72

P

process, 50
process measurement, 50
project, 51

Q

quality assessment cycle, 74
quality assessment life-cycle, 74
quality assessment metamodel, 84
quality assessment model, 81, 82
quality assessment modelling, 76
quality factor, 49
quality indicator, 119
quality issue, 82, 113
quality model, 35, 49
quality profile, 150

S

scale, 49
software product, 51
software project, 67
software ecosystem, 41
stakeholder, 51

U

unit of measurement, 50

V

value, 50