REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université de Batna 2

Faculté des Mathématiques et de l'Informatique

Département d'Informatique

THESE

En vue de l'obtention du diplôme de

Doctorat LMD en Informatique

Spécialité : ***Systèmes Informatiques (SI)***

Présentée par

Salim KADRI

Contrôle de qualité des architectures logicielles à base de composants : Cas d'étude

---

Soutenue publiquement le : 03/07/2022

Jury :

Pr. Saber BENHARZALLAH	*Professeur*	*Université de Batna 2*	*Président*
Pr. Sofiane AOUAG	*Professeur*	*Université de Batna 2*	*Rapporteur*
Pr. Laid KAHLOUL	*Professeur*	*Université de Biskra*	*Examinateur*
Pr. Hammadi BENNOUI	*Professeur*	*Université de Biskra*	*Examinateur*
Pr. Rachid SEGHIR	*Professeur*	*Université de Batna 2*	*Examinateur*
Pr. Djalal HEDJAZI	*Professeur*	*Université de Batna 2*	*Invité*

PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

Ministry of Higher Education and Scientific Research

University of Batna 2

Faculty of Mathematics and Computer Science

Computer Science Department

DISSERTATION

In order to obtain the diploma of

LMD Ph.D. in Computer Science

Specialty: Informatics Systems (IS)

Presented by

Salim KADRI

Quality control of component-based software architectures: Case study

---

Publicly defended on July 03, 2022

Jury :

Pr. Saber BENHARZALLAH	*Professor*	*Université de Batna 2*	*President*
Pr. Sofiane AOUAG	*Professor*	*Université de Batna 2*	*Reporter*
Pr. Laid KAHLOUL	*Professor*	*Université de Biskra*	*Examiner*
Pr. Hammadi BENNOUI	*Professor*	*Université de Biskra*	*Examiner*
Pr. Rachid SEGHIR	*Professor*	*Université de Batna 2*	*Examiner*
Pr. Djalal HEDJAZI	*Professor*	*Université de Batna 2*	*Guest*

Abstract

For many decades, software quality has been considered the key ingredient for business success for organizations. In this context, software architecture has been held as the appropriate level to deal with quality requirements (quality attributes). Various methods have been proposed to evaluate software architecture; however, according to the literature study we had performed, we found that these methods suffer from many drawbacks and shortcomings. In this dissertation, we have attempted to overcome these shortcomings by proposing a new evaluation methodology within a multi-service evaluation framework called MS-QuAAF (**M**ulti-**S**ervice - **Qu**antitative **A**rchitecture **A**ssessment **F**ramework). The latter consists of two primary modules. The first module proposes a new concept called facet projection to reduce the complexity of the target architectures by downsizing the relevant meta-models. The second module provides a set of generic metrics applied to the target architectures through three assessment services. The first service allows assessing the defectiveness of architecture at the design stage. The second service is used to assess architecture at the implementation stage through a newly proposed method called the responsibility decomposition analysis. The third service is dedicated to finalizing the evaluation effort and generating the final assessment report.

The experimental evaluation that we had conducted through two cases study allowed us to answer the confronted research questions, thus many findings and contributions have been achieved. Contrarily to most methods, MS-QuAAF can provide a continuous evaluation that covers two main development stages, which are the design and implementation stages. It allows through its generic metrics the evaluation of any inputted quality attribute and the detection of architecture deviations. Furthermore, the framework is capable of steering architects during the development process to detect irregularities and hence improve architecture quality.

Keywords. Software Quality; Software Architecture; Component-based Software; Quality Attributes; Non-Functional Requirements; Quality evaluation; Quality metrics; Architecture Defects.

Résumé

Depuis plusieurs décennies, la qualité des logiciels a été considérée comme le constituant clé du succès commercial des organisations. Les architectures logicielles ont été jugées comme le niveau approprié pour traiter les exigences de qualité des utilisateurs (attributs de qualité). Une multitude de méthodes a été proposée pour évaluer l'architecture logicielle. Cependant, selon l'étude de la littérature que nous avions effectuée, nous avons constaté que ces méthodes souffrent de nombreuse lacunes. Dans cette thèse, notre objectif est de surmonter ces lacunes en proposant une nouvelle méthodologie d'évaluation à l'aide d'un framework d'évaluation multiservice appelé MS-QuAAF. Ce framework se compose de deux modules principaux. Le premier module propose un nouveau concept appelé projection de facettes pour réduire la complexité d'une architecture cible en réduisant la taille de ses méta-modèles. Le deuxième module fournit un ensemble de métriques génériques appliquées aux architectures à travers trois services d'évaluation. Le premier service permet d'évaluer la défectuosité de l'architecture durant l'étape de conception. Le deuxième service est utilisé pour évaluer l'architecture à l'étape d'implémentation grâce à une méthode proposée appelée analyse de décomposition des responsabilités. Le troisième service est dédié à la finalisation de l'évaluation et à la production du rapport final.

L'évaluation expérimentale que nous avions conduit à travers deux cas d'études nous a permis de répondre aux questions de recherche confrontées, ainsi de nombreuses conclusions et contributions ont été obtenues. Contrairement à la plupart des méthodes proposées, MS-QuAAF peut fournir une évaluation continue qui couvre deux étapes principales de développement : l'étape de conception et l'étape d'implémentation. Le framework permet à travers ses métriques génériques, l'évaluation de tous les attributs de qualité et la détection des déviations d'architecture. De plus, il est capable de guider les architectes pendant le processus de développement pour détecter les irrégularités et améliorer la qualité de l'architecture.

Mots clés. Qualité du logiciel ; Architecture logicielle; Logiciel à base de composants ; Attributs de qualité ; Exigences non fonctionnelles; Évaluation de la qualité ; Métriques de qualité ; Défauts architecturaux.

ملخـــص

لسنوات عديدة، اعتبرت جودة البرمجيات العنصر الرئيسي لنجاح للمؤسسات في أعمالها. في هذا السياق، تعتبر بنية هاته البرمجيات على أنها المستوى المناسب للتعامل مع متطلبات الجودة. مع زيادة الطلب من طرف المستخدمين على تلبية هذه المتطلبات، يزداد حجم وتعقيد بنية البرامج. لقد أدرك الباحثون في هذا المجال أن تحقيق سمات الجودة مقيد ببنية برمجيات هذه الأنظمة. تم اقتراح عدد كبير من الطرق لتقييم بنية البرامج، خاصة في المراحل الأولى من عملية التطوير. ومع ذلك، ووفقًا للدراسة التي أجريناها، وجدنا أن هذه الأساليب تعاني من العديد من العيوب.

في هذه الأطروحة، حاولنا التغلب على النقائص المسجلة من خلال اقتراح منهجية تقييم جديدة ضمن إطار تقييم متعدد الخدمات يسمى MS-QuAAF. يتكون هذا الأخير من وحدتين أساسيتين .تقترح الوحدة الأولى طريقة جديدة لتقليل تعقيد البنى من خلال تقليص حجم نماذجها الوصفية . أما الوحدة الثانية فتوفر مجموعة من المقاييس العامة والتي تطبق على البنى من خلال ثلاث خدمات تقييم .تتيح الخدمة الأولى تقييم عيوب البنى في مرحلة التصميم .يتم استخدام الخدمة الثانية لتقييم البنى في مرحلة التنفيذ. الخدمة الثالثة مخصصة لإعداد تقرير التقييم النهائي.

التقييم التجريبي الذي أجريناه من خلال دراسة حالتين سمح لنا بالإجابة على أسئلة البحث التي تم طرحها، وبذلك تم تحقيق العديد من النتائج والمساهمات .على عكس معظم الطرق الأخرى، يمكن أن يوفر MS-QuAAF تقييمًا مستمرًا يغطي مرحلتين رئيسيتين من مراحل التطوير، وهما مراحل التصميم والتنفيذ . كما يسمح أيضا بتقييم كل سمات الجودة من خلال مقاييسه العامة المقترحة. علاوة على ذلك، فهو يشتمل على ميزة تسمح باكتشاف تآكل البنى .على الرغم من مصادفتنا بعض المشكلات والقيود أثناء التقييم، إلا أننا نعتقد أن إطار العمل المقترح في هذه الأطروحة قادر على توجيه المهندسين أثناء عملية التطوير لاكتشاف المخالفات والنقائص وتحسين جودة البنية من خلال خدماته.

كلمات مفتاحية .جودة البرمجيات؛ بنية البرمجيات؛ البرامج القائمة على المكونات؛ علامات الجودة؛ متطلبات غير وظيفية؛ تقييم الجودة؛ مقاييس الجودة؛ عيوب البنى.

Dedication

I dedicate this dissertation to the memory of my father, a man like no other. To my precious mother, who has been my first source of inspiration to pursue my Ph.D. studies. To my beloved wife for her constant support and encouragement throughout this dissertation. To my admired son JAD, my little hero. To my dearest brothers and sisters, who have always been by my side. Finally, God bless all the people who helped us accomplish this dissertation.

Acknowledgments

As I type this, I have spent three years and two months working on this dissertation. I would take this opportunity to express my sincere thanks and feelings of gratitude to my supervisor, Sofiane AOUAG, for his vital support and guidance. I would also like to offer special thanks to the members of the jury, professors Saber BENHARZALLAH as a president, Laid KAHLOUL as an examiner, Hammadi BENNOUI as an examiner, Rachid SEGHIR as an examiner, and Djalal HEDJAZI as a guest for accepting to examine my work. It is truly a great honor for me.

Contents

Chapter 1: Introduction ... 1
1.1 Context ... 1
1.1.1 Component-based software engineering ... 1
1.1.2 Software architecture ... 2
1.1.3 Software quality ... 2
1.2 Motivation for architecture evaluation ... 3
1.3 The problematic of architecture evaluation ... 4
1.4 Dissertation proposal and contributions ... 5
1.5 Dissertation outline ... 6
Chapter 2: Component-based software engineering and software architecture ... 9
2.1 Component-based software engineering ... 9
2.2 Characteristics of CBSE ... 11
2.3 Component-based software life cycle ... 12
2.3.1 Traditional software engineering paradigms ... 12
2.3.1.1 The waterfall paradigm ... 12
2.3.1.2 The incremental development paradigm ... 14
2.3.1.3 The spiral development paradigm ... 14
2.3.2 Advanced software engineering paradigms ... 15
2.3.2.1 The Agile development paradigm ... 15
2.3.2.2 Component-based software paradigm ... 16
2.4 Software architecture and components ... 18
2.4.1 Architecture ... 18
2.4.2 Components ... 19
2.4.2.1 Basic elements of components ... 20
2.4.2.1.1 Interfaces ... 20
2.4.2.1.2 Implementation ... 21
2.4.2.1.3 Specification ... 21
2.4.3 Connectors ... 22
2.4.4 Architecture modeling ... 23
2.4.5 Architecture Description Languages (ADLs) ... 24
2.5 Architecture analysis ... 25
2.5.1 Architectural characteristics to be analyzed ... 27
2.5.2 Analysis techniques ... 27

2.6 Conclusion ... 28
Chapter 3: Software quality ... 31
3.1 What is software quality? ... 31
3.1.1 The quality according to Shewhart ... 32
3.1.2 The quality according to Deming ... 32
3.1.3 Quality according to Juran ... 33
3.1.4 Quality according to Crosby ... 34
3.1.5 Quality according to Ishikawa ... 35
3.1.6 Quality according to IEEE Std.730-2014 ... 35
3.1.7 Comparison and contrast ... 35
3.1.8 Our perspective on software quality ... 37
3.2 Software quality management ... 37
3.2.1 Software product stakeholders ... 38
3.2.2 Total Quality Management ... 38
3.2.3 Software Quality Assurance ... 39
3.2.4 Software quality control ... 39
3.2.5 SQA Vs SQC ... 40
3.3 Error, fault (defect), failure ... 41
3.3.1 Errors ... 41
3.3.2 Software defects ... 42
3.3.3 Software failure ... 43
3.4 Quality models ... 44
3.4.1 The McCall's quality model ... 44
3.4.2 The Boehm's quality model ... 46
3.4.3 The Dromey's quality model ... 47
3.4.4 The FURPS quality model ... 48
3.4.5 The ISO/IEC 25010:2011 Quality Models ... 49
3.4.6 Analysis and discussion ... 51
3.4.7 MS-QuAAF and Quality models ... 53
3.5 Conclusion ... 53
Chapter 4: Software architecture analysis and evaluation methods ... 56
4.1 Motivation for architecture evaluation ... 56
4.2 Classification criteria for evaluation methods ... 57
4.3 Evaluation methods ... 59
4.3.1 Scenario-based methods ... 59
4.3.1.1 Scenario-based Architecture Analysis Method (SAAM) ... 60

4.3.1.2 Extending SAAM by integrating in the domain (ESAAMI) ... 62
4.3.1.3 Software Architecture Analysis Method for Evolution and Reusability (SAAMER) ... 63
4.3.1.4 The Architecture Trade-off Analysis Method (ATAM) ... 65
4.3.1.5 Scenario based architecture reengineering (SBAR) ... 67
4.3.1.6 Cost-Benefit Analysis Method (CBAM) ... 68
4.3.1.7 Architecture-Level Modifiability Analysis (ALMA) ... 69
4.3.1.8 Active Reviews for Intermediate Design (ARID) ... 70
4.3.2 Goal decomposition analysis methods ... 73
4.3.2.1 The NFR framework ... 73
4.3.2.2 A goal-oriented simulation approach for cloud-based system architecture ... 74
4.3.2.3 A quantitative assessment method for analyzing safety and security using the NFR approach ... 76
4.3.2.4 A quantitative architecture evaluation using the goal decomposition framework and Archimate ... 77
4.3.3 Metric-based evaluation methods (metrics suites) ... 80
4.3.3.1 Traditional metrics ... 81
4.3.3.1.1 Size related metrics ... 82
4.3.3.1.2 Complexity metrics ... 82
4.3.3.2 Object-oriented metrics ... 83
4.3.3.2.1 Metrics for Object-Oriented Software Engineering (MOOSE) ... 83
4.3.3.2.2 Chen's object-oriented metrics ... 84
4.3.3.2.3 Metrics for Object Oriented Design (MOOD) ... 85
4.3.3.2.4 A Hierarchical Quality Model for Object-Oriented Design (QMOOD) ... 86
4.3.3.2.5 Li's object-oriented metric suite ... 87
4.3.3.2.6 Choi's Component-based metrics ... 87
4.3.4 A myriad of other approaches to assess and predict quality attributes ... 89
4.4 A concluding discussion about the presented evaluation methods ... 89
4.5 Conclusion ... 91
Chapter 5: The framework MS-QuAAF for monitoring and evaluating architecture quality: foundation and evaluation methodology ... 94
5.1 Motivations and rationale behind MS-QuAAF ... 94
5.2 An introduction to the MS-QuAAF's evaluation methodology ... 96
5.2.1 The ISO/IEC/IEEE 42030:2019 architecture evaluation framework ... 98
5.2.2 The compliance of MS-QuAAF with The ISO/IEC/IEEE 42030:2019 ... 99
5.3 The facet projector module ... 101
5.3.1 Architecture facet ... 101

5.3.2 The NFR catalog 101
5.3.3 The anatomy of an architecture facet 102
5.3.4 Facets Vs AOSD's aspects 104
5.3.5 Facet projection 105
5.3.6 An illustrative example of facet projection 107
5.3.7 Expected properties of the generated facets 112
5.4 Conclusion 112
Chapter 6: The MS-QuAAF's evaluation services and metrics 115
6.1 The evaluation effort 115
6.2 The quality evaluator module 117
6.2.1 Rules Defectiveness Analysis service (RDA) 117
6.2.1.1 Macro-groups 118
6.2.1.2 Calculating architecture defectiveness metrics 118
6.2.1.3 Macro-groups weightage 121
6.2.2 Responsibilities Tree Assessment service (RTA) 122
6.2.2.1 The Responsibilities Satisfaction Tree (RST) 123
6.2.2.2 Quantitative evaluation using RST 125
6.2.3 Late-assessment and final analysis service (LAFA) 129
6.2.3.1 Evaluation model 129
6.2.3.2 Assessing quality using Euclidean distance as a closeness measure 130
6.3 Conclusion 132
Chapter 7: An experimental evaluation of the framework MS-QuAAF: A case study 135
7.1 The experimental evaluation protocol 135
7.1.1 The research methodology 136
7.1.1.1 Research purpose 136
7.1.1.2 Case study design 137
7.1.2 Target architecture 138
7.1.3 Groups configuration 139
7.1.4 The MS-QuAAF tool prototype 140
7.2 The Experimental evaluation process 142
7.2.1 Architecture specification 142
7.2.1.1 Specifying SA1 using Alloy 142
7.2.1.2 Specifying SA2 using OCL and eclipse Ecore 144
7.2.2 Design-time evaluation 148
7.2.3 NFR responsibilities evaluation (implementation-time) 154
7.2.4 Final assessment report 158

7.3 Answering the research questions ... 162
7.4 Threats to validity ... 163
7.5 Conclusion ... 164
Chapter 8: Conclusion ... 166
8.1 Research aims and overall findings ... 166
8.2 Contributions ... 167
8.2.1 Architecture modeling and slicing contributions ... 167
8.2.2 Architecture evaluation contributions ... 168
8.2.3 Applicative contribution ... 169
8.3 Limitations and perspectives ... 169
List of published papers ... 172
Bibliography ... 173
Appendix A ... 185
Appendix B ... 189
Appendix C ... 195

List of figures

Figure 2.1. Concurrent CBSD process (Tiwari & Kumar, 2020)..... 10
Figure 2.2. The Waterfall model..... 13
Figure 2.3. A V development model adapted to CBS paradigm..... 18
Figure 2.4. Component's interfaces (Sommerville, 2011)..... 22
Figure 3.1. The causality relationship between error, fault, and failure..... 41
Figure 3.2. Software errors, defect, and failure..... 44
Figure 3.3. The McCall's quality triangle..... 45
Figure 3.4. The McCall's hierarchical quality model..... 46
Figure 3.5. The Boehm's quality model..... 47
Figure 3.6. The Dromey's quality model..... 48
Figure 3.7. The FURPS's quality model..... 49
Figure 3.8. A quality in use model (ISO/IEC 25010:2011)..... 51
Figure 3.9. A product quality model (ISO/IEC 25010:2011)..... 51
Figure 4.1. The relationship between software architecture and business goals..... 57
Figure 4.2. Classification criteria for evaluation methods..... 60
Figure 4.3. ESAAMI approach..... 63
Figure 4.4. SBAR activities..... 68
Figure 4.5. An example of performance and security SIGs..... 74
Figure 4.6. A quantitative SIG using the SV mapping scheme..... 76
Figure 4.7. Quantitative architecture evaluation using the NFR goal tree..... 78
Figure 4.8. A subset of design-based metrics..... 81
Figure 5.1. A simplified conceptual model of MS-QuAAF..... 98
Figure 5.2. The architecture evaluation context..... 99
Figure 5.3. The anatomy of an architecture facet..... 104
Figure 5.4. The facet projector input/output..... 106
Figure 5.5. The graphical representation of the security facet..... 111
Figure 6.1. MS-QuAAF evaluation effort..... 116
Figure 6.2. The Responsibilities Satisfaction Tree (RST)..... 125
Figure 6.3. An example of a quantitative evaluation using RST..... 129
Figure 7.1. The structure of our embedded single-case study..... 138
Figure 7.2. A conceptual representation of the target architectures SA1 and SA2..... 140
Figure 7.3. The main window of the MS-QuAAF tool prototype..... 141
Figure 7.4. An Entity/Relation diagram of the NFR catalog..... 142
Figure 7.5. The OCLinEcore plugin used to embed OCL invariants within EMF. 145
Figure 7.6. Extracting a facet through the Facet projector GUI..... 150

Figure 7.7. Selecting the target quality attribute 152
Figure 7.8. Calculating QuARF and QuARD using the MS-QuAAF tool 153
Figure 7.9. A graphical representation of architecture defectiveness of AF1 and AF5 154
Figure 7.10. Overall architectural defects 154
Figure 7.11. Calculating the RSI metric of the performance attribute 156
Figure 7.12. Calculating RSI of performance using the MS-QuAAF tool 158
Figure 7.13. The ED between extensibility and performance vectors and the optimum vectors 160

List of Tables

Table 3.1. The comparison between the quality pioneers' viewpoints 36
Table 3.2. A comparison between software quality assurance and software quality control..... 41
Table 3.3. comparison between the presented quality models 52
Table 4.1. Scenario-based architecture evaluation methods 72
Table 4.2. A comparison between the presented goal decomposition analysis methods..... 80
Table 4.3. A comparison between traditional and OO metrics..... 88
Table 4.4. A comparison between all the presented evaluation methods 91
Table 6.1. MS-QuAAF's Services 116
Table 6.2. The Saaty scale..... 121
Table 6.3. MS-QuAAF metrics suite..... 132
Table 7.1. Development groups' configuration..... 140
Table 7.2. Quality attribute, number of rules, and specification language of SA1 and SA2 143
Table 7.3. Examples of SA1 and SA2's general rules with their importance level..... 148
Table 7.4. The initial Pairwise comparison matrix 150
Table 7.5. The normalized comparison matrix..... 150
Table 7.6. Calculating QuARF and QuARD of SA1 152
Table 7.7. Calculating QuARF and QuARD of SA2 152
Table 7.8. The obtained RSIs of SA1 and SA2..... 156
Table 7.9. The evaluation matrix X1 158
Table 7.10. The evaluation matrix X2 158
Table 7.11. ClosH results 159
Table 7.12. ClosV results 159

List of abbreviations

ADL: Architecture Description Language.
AE: Architecture Evaluation.
AF: Architecture Facet.
CBSD: Component-based Software Development.
CBSE: Component-based Software Engineering.
ClosH: Horizontal Closeness.
ClosV: Vertical Closeness.
LAFA: Late-assessment and Final Analysis.
MG: Macro-group.
MGF: Macro-group Fulfillment.
MS-QuAAF: Multi-**S**ervice - Quantitative Architecture Assessment Framework.
NFR: Non-Functional Requirements.
OVD: Overall Defectiveness.
OVF: Overall Fulfillment.
QA: Quality Attribute.
QuARD: Quality Attribute Rules Defectiveness.
QuARF: Quality Attribute Rules Fulfillment.
RDA: Rules Defectiveness Analysis.
RQ: Research Question.
RSI: Responsibilities Satisfaction Indicator.
RTA: Responsibilities Tree Analysis.
SA: Software Architecture.
SDLC: Software Development Life Cycle.

Chapter 1

Introduction

In this introductory chapter, we will briefly present the research context of this dissertation, the motivation for architecture evaluation, research aims and dissertation proposal, contributions, and the organization of the dissertation.

Contents

1.1 Context 1
1.2 Motivation for architecture evaluation 3
1.3 The problematic of architecture evaluation 4
1.4 Dissertation proposal and contributions 5
1.5 Dissertation outline 6

1.1 Context

1.1.1 Component-based software engineering

In the late 1990s, component-based software engineering (CBSD) has emerged as a new approach and trend to develop software systems based on software components reuse (Sommerville, 2011). The main motivation and rationale behind this development approach is the failure of the Object-Oriented paradigm to meet reusability expectations. In this context, distributing objects as reusable components is practically unrealizable because classes are too detailed, specific, and require a thorough understanding of their source code. Contrastingly, components are deployable artifacts of a higher abstraction level, specified by opened interfaces, and the implementation is hidden from other components. Therefore, component-based software engineering aims at assembling large software systems from new and pre-existing components through well-defined interfaces (Gaedke & Rehse, 2000). The embraced philosophy is to reuse pre-constructed software components instead of developing them from scratch. Hence, a component is developed only once and subsequently reused in various applications rather than reconstructing it every time. Accordingly, CBSD is characterized as a sub-discipline of software engineering particularized by reusability, composability, maintainability, adaptability, shorter development cycle, and improved quality properties.

The primary goals of CBSD are to reduce the development effort and cost, shorten the software development cycle by deploying and delivering software applications more quickly, increase reliability, and facilitate maintenance by replacing, updating, and upgrading the concerned components without affecting the whole system. However, certain issues have been addressed

during building component-based systems, such as the trustworthiness of the deployed components, integration and interaction issues, and testing issues. These concerns can affect the quality of the developed components, and thus the overall system quality.

1.1.2 Software architecture

Software architecture has emerged as a mature and central sub-discipline of software engineering after several years of the software crisis in the mid-1970s (Kruchten et al., 2009). It can be defined as a set of significant decisions about the organization of a software system. These decisions include the selection of structural elements and their interfaces, the collaboration among those elements (the behavior), the composition of these structural elements into large subsystems, and the architectural styles that guide this organization (Bass et al., 2012; Kruchten et al., 2009; Mary & David, 1996). The architecture is generally constructed based on architectural decisions made in the early stages of the development process to achieve functional and non-functional requirements (*called also quality attributes*) (Falessi et al., 2011). However, the ability of a software system to fulfill its assigned functional responsibilities does not imply that the predefined quality attributes (performance, security, extensibility, etc.) are met. For instance, a system can deliver efficiently correct results but it does not satisfy the security requirement by letting confidential data be exposed to malicious users, or it takes too long to deliver these results. The solution to these quality concerns is purely architectural. In this context, architects should take the right architectural decisions to promote each one of the stakeholders' (clients, developers, etc.) quality attributes. The more demanding the stakeholders are, the more complex and critical the architecture is. As a result, delivering software of high quality depends on implementing faithfully the architecture designed based on the architectural decisions taken earlier. Contrarily, violating these decisions by designers and developers throughout the development process may influence negatively the quality of the software and jeopardize its business goals.

As the complexity of software architecture increases, the effort of architecture analysis and quality assessment increases as well. Consequently, architecture models are used as an abstraction of a software system to reduce its complexity by representing only the aspects of interest to the task at hand. These models are expressed using architectural modeling notations (formal, semi-formal, etc.) to capture the architectural decisions taken to build an architecture. In this Dissertation, we focus only on the decisions made to satisfy stakeholders' quality attributes. We have proposed a new concept called architectural facets to reduce the complexity of architecture. Each facet is mapped to one quality attribute by encompassing only the decisions made to promote this attribute (e.g., performance). Subsequently, these facets are analyzed and assessed one by one to evaluate the overall architecture.

1.1.3 Software quality

"*What is software quality?*" is a kind of question that generates different answers depending on whom we asking (consumers or producers of the system), type of software system, business

and market environment, and so forth. From the point of view of producers, a product is judged of high quality if it conforms to the specified requirements. On the other hand, a customer's viewpoint is when the product meets his/her needs and expectations. In this connection, the pioneers of quality are divided into two major camps, the *Conformance to Requirements* camp and the *Customer Satisfaction* camp. However, they share two main similarities. The former is that quality is the responsibility of the senior management. The latter is that individuals involved in the production process must get training and continuous education to improve quality.

In this Dissertation, our viewpoint is tight-coupled with the Non-Functional Requirements (NFRs), which are characteristics of a software system called often quality attributes (QAs). These attributes represent measurable (or testable) properties of a software system used to designate its goodness and how well it satisfies the needs of its stakeholders (Bass et al., 2012). Contrarily to functional requirements that refer to the purpose of the system or the intended functions supposed to be provided by the system, quality attributes permit to determine how the system is supposed to be. For instance, the system should be maintainable, secure, and available. Therefore, *maintainability*, *security*, and *availability* represent the system's quality attributes.

Accordingly, our definition to software quality is the degree of achievement (satisfaction) of the competing quality attributes desired by all stakeholders and not just customers. For instance, architects and developers are interested in portability and modifiability, whereas customers are interested in efficiency and usability. However, the achievement of quality attributes is correlated with the conformance to the quality requirements specified as architectural decisions at the early stages of the development process.

1.2 Motivation for architecture evaluation

Nowadays, the software marketplace is highly competitive, which means that delivering high-quality software is not only an obligatory but also a differentiator factor for companies. Critical sectors that depend heavily on software, such as homeland security and air control traffic expect to deploy error-free systems that are supposed to fulfill their quality requirement flawlessly. Therefore, the slight difference between the quality of a software product and its competitors is considered decisive in this highly demanding market. Furthermore, poor software quality is responsible for severe business and safety disasters (Carrozza et al., 2018). Companies must boost up software quality perception and integrate it profoundly into their development process. They need tools and frameworks to support and perform architecture assessment even if it cost them the changing of their cultural, technical, and organizational reasoning. The lack of knowledge about software quality inside an organization is one of the main reasons that cause software defects. Therefore, quality must be perceived by the top management as well as the staff involved in the development of software products.

Since it has emerged in the last decade of the 20th century as a sub-discipline of software engineering, software architecture has been considered as the appropriate level to deal with quality attributes (Dobrica & Niemela, 2002). Hence, it has attracted the attention of researchers, as well as practitioners since the size and complexity of software systems have been augmented tremendously along with the increased demand of stakeholders' to satisfy their quality requirements. They have realized (researcher and practitioners) that high-level design description is the key to understanding and managing complex software systems. Moreover, they recognized that the achievement of quality attributes is constrained by the SA of these systems and the architectural decisions made at this level (Babar et al., 2004).

1.3 The problematic of architecture evaluation

Undoubtedly, SA plays a major role in the achievement of the desired quality attributes; therefore, analyzing and assessing SA against those attributes of concern is very important as early as possible. In this context, the main goal of SA evaluation is to detect architectural defects and liabilities that may degrade the quality of architecture, and thus determining the extent to which architectural decisions meet the quality requirements. There were a myriad of evaluation methods proposed to assess software architecture. These methods can be divided into two basic classes, a qualitative evaluation, and a quantitative evaluation (Abowd et al., 1997; Dobrica & Niemela, 2002). The former uses questioning techniques to generate qualitative questions to be asked of an architecture. These techniques include questionnaires, checklists, and scenarios. The latter suggests quantitative measurement to evaluate an architecture using, for example, metrics, simulations, and experimentations. Inside each one of these evaluation classes, methods can be categorized according to many criteria, such as the target evaluation stage and the number of supported quality attributes.

Indeed qualitative evaluation is thought to be applicable to assess any given architecture quality (Abowd et al., 1997; Bass et al., 2012); however, it lacks statistical significance. Questioning techniques like questionnaires and checklists are mostly based on the evaluators' perspective and subjectivity, which may decrease the evaluation accuracy and objectivity. Additionally, the results of scenario-based analysis depend on the selection of the scenarios and their relevance for identifying architecture's weaknesses. In this context, there is no fixed number of scenarios to guarantee that the evaluation analysis is meaningful (Dobrica & Niemela, 2002). On the other hand, the essence of measuring techniques is to deliver assessors with quantitative results and views that can reflect the state of the architecture quality more accurately. However, this type of evaluation is addressed to answer specific questions, and thus evaluating specific quality attributes. Additionally, defining metrics for some attributes, such as security, and usability have proven difficult to develop (Bachmann et al., 2005). Predominantly, the literature study that we had performed to analyze the proposed methods allowed us to discover the following drawbacks:

- The evaluation is performed, either at the design time or implementation stage. Therefore, there is no such feature as continuous evaluation.
- There is no conformance analysis between the prescribed architecture (the realized one) and the descriptive one (the intended one). As a result, architecture deviation analysis is not included within these methods.
- Most methods support the evaluation of one or two quality attributes, especially quantitative methods. Very few methods that can support the evaluation of a large set of quality attributes.
- Qualitative methods lack statistical significance.
- Metric-based methods assess only some internal characteristics (especially complexity) correlated with a small set of quality attributes (maintainability and performance).
- Lack of tool support.

1.4 Dissertation proposal and contributions

In this Dissertation, our main research aim is to propose an evaluation methodology that attempts to overcome the above-stated shortcomings. Accordingly, we have proposed a new evaluation framework called MS-QuAAF (**M**ulti-**S**ervice - **Qu**antitative **A**rchitecture **A**ssessment **F**ramework), a generic metric-based framework for evaluating software architecture continuously at the design and implementation stages (Kadri et al., 2021b). The framework defines a suite of generic evaluation metrics to help evaluators in:

- Assessing any quality attribute inputted into the framework.
- Assessing architecture throughout two main stages of the development process (design and implementation).
- Estimating architecture defectiveness and detecting potential deviations.
- Evaluating the achievement of the NFR responsibilities assigned to promote quality attributes.
- Making decisions about the architecture (e.g., validating or disapproving the design).

MS-QuAAF performs architecture evaluation through two main modules. The first module implements the concept of facet projection to extract from architecture meta-models only information of interest to the evaluation task. The second module proposes seven metrics applied to the target architectures through three assessment services. The framework allows quality evaluation and monitoring throughout two main stages of the development process, more specifically, after the accomplishment of the design and implementation stages. The former evaluation allows identifying architecture defects, which helps architects to fix design flaws according to the early architecture specification. The latter evaluation assesses the

fulfillment of the NFR responsibilities prescribed to promote stakeholders' quality attributes. As a result, incorporating these two types of assessment in the same framework allows a) continuous architecture evaluation and monitoring, b) identifying whether a poor architecture quality is caused by rules infringement at the design stage, implementation stage, or both of them, and c) improving the quality by adjusting the architecture in accordance with the architecture specification. Additionally, the evaluation is performed quantitatively through a set of metrics to provide architects and evaluators with quantitative views that reflect the state of architecture quality more accurately.

During the development of MS-QuAAF, we have confronted with the following research questions:

- **RQ1:** Is the proposed framework capable of calculating the defectiveness of the designed architecture?
- **RQ2:** Is the proposed framework capable of estimating the satisfaction of NFR responsibilities of the implemented architecture?
- **RQ3:** Are the proposed metrics capable of deducing architecture deviations?
- **RQ4:** Does the proposed framework help in enhancing architecture quality?

After we had conducted a case study to answer these questions, we succeeded to achieve the following contributions:

- A new concept called facet projection is proposed to reduce the complexity of architecture in order to facilitate its assessment and monitoring. The latter consists of extracting from large meta-models only information of interest to the evaluation task at hand.
- The evaluation proposed by the framework is independent of quality attributes and their relevant quality models, which makes it a generic framework. The independency is promoted by the proposed generic metrics that direct the ability of the framework to evaluate any inputted quality attribute.
- The framework promotes continuous evaluation by covering two crucial stages of the development process, which are the design and the implementation stages. Additionally, the evaluation can be performed after major maintenance activities.
- A new concept called NFR responsibilities is proposed within the framework to evaluate the implemented architecture. For each quality attribute, NFR responsibilities are decomposed iteratively to construct a weighted analysis tree used to assess the fulfillment of these responsibilities.

1.5 Dissertation outline

The dissertation is organized as follows.

- **Chapter 1** introduces the context of this dissertation, which is the component-based software architecture quality control and monitoring. Moreover, we have introduced the motivation for architecture evaluation, problematic of architecture evaluation, the dissertation proposal, and the research question.
- **Chapter 2** depicts two sub-disciplines of software engineering and the relationship between them, which are component-based software engineering and software architecture. Architecture modeling, architecture description languages, and architecture analysis are also depicted in this chapter.
- **Chapter 3** describes the concept of quality according to the most-known quality management pioneers, software quality management techniques, software defects and failure, and the most-known quality models proposed to provide standardized tools to measure software quality.
- **Chapter 4** constitutes a state-of-the-art study related to the topic of this dissertation. We have presented a large set of evaluation techniques proposed to evaluate software architecture. The analysis of these methods and the comparison between them gave us a clear view of the domain of software architecture evaluation, although its diversity and complexity. Accordingly, we extracted from these evaluation techniques the main strengths and weaknesses to develop a new framework that encompasses new features within a new generic evaluation perspective.
- **Chapter 5** presents firstly the rationale behind proposing the evaluation methodology within MS-QuAAF. Second, the framework's methodology and its foundation are presented in a nutshell. Third, the concept of architecture facets, model projection, and facet projection defined within the facet projector module are depicted.
- **Chapter 6** details the evaluation effort performed through the quality evaluator module. The latter consists of three evaluation services each of which defines a set of generic metrics calculated mathematically through sequences of equations.
- **Chapter 7** provides a concrete implementation of each evaluation service through an experimental evaluation, in which two case studies will be treated and discussed. The implementation will illustrate how to use the proposed metrics to assess software architecture during the development process. Finally, the research questions are answered in this chapter.
- **Chapter 8** concludes this dissertation by summarizing the key finding of this research by answering the research questions. Additionally, it discusses the contributions of the proposed approach, its limitation and weaknesses, and perspectives and future work.

Chapter 2

Component-based software engineering and software architecture

In this chapter, pertinent concepts and generalities from two related and prominent sub-disciplines of software engineering will be addressed before tackling the main topic of this dissertation, which is controlling architecture quality and evolution. These concepts may be used implicitly and explicitly throughout the quality control process. The first sub-discipline is called component-based software engineering (CBSE). The latter is the software architecture sub-discipline.

Accordingly, this chapter is organized as follows. The first section is an introduction to CBSE, which is a development model and an approach to building software architecture from pre-existed components in order to improve quality, decreasing the development time, and facilitating maintenance activities. The second section depicts the main characteristics of CBSE that may incite software architects to adopt this development model. The third section depicts traditional and advanced software engineering paradigms including CBSE. The fourth section illustrates software architecture and its fundamental constituents, which are components and connectors. Additionally, we will present architecture modeling and its notations, and Architecture Description Languages (ADL) used to specify software architectures. The fifth section depicts architecture analysis and its essential techniques, which considered central for controlling software architecture quality during its life cycle.

Contents

2.1 Component-based software engineering 9
2.2 Characteristics of CBSE 11
2.3 Component-based software life cycle 12
2.4 Software architecture and components 18
2.5 Architecture analysis 25
2.6 Conclusion 28

2.1 Component-based software engineering

Component-based software engineering has emerged in the late 1990s as an approach to software systems development based on software components reuse (Sommerville, 2011). The failure of object-oriented development (OOD) to meet the reusability expectations is the main

motivation for creating CBSE. In this context, classes are too specific, detailed, and require in-depth knowledge of source code to use them, which made distributing objects as reusable components practically infeasible. As a result, OOD is used to construct monolithic applications, in which objects are almost never sold, bought, or deployed (C. Szyperski et al., 2002). On the other hand, components are deployable artifacts of higher abstraction level specified by interfaces, generally larger than objects, and all implementation details are hidden from other components. According to (Gaedke & Rehse, 2000), "CBSE aims at assembling large software systems from previously developed components (which in turn can be constructed from other components)". More specifically, applications are constructed by composing reusable new and pre-existing components through well-defined opened interfaces. Consequently, Many adopters of OOD moved to the camp of CBSD (Component-based software development) to improve reusability and enhance the overall architecture quality (Pree, 1997).

The philosophy of CBSE is to reuse pre-constructed software components instead of developing them from scratch. The fundamental idea is to develop a component only once and reuse it in various applications rather than reconstruct it every time (Tiwari & Kumar, 2020). The CBSE process consists of defining, implementing, and integrating (composing) independent loosely coupled components into larger and complex systems. This indicates that CBSD is dedicated to developing specific as well as generalized components by following four processes, which are developing new components, selecting pre-constructed components from the repository, assembling components, and control and management (figure 2.1). These processes can be performed concurrently, in which feedbacks methods must be defined to address and fix the emerged problems with each process. Similarly, (Jacobson et al., 1997) defined another four parallel processes paradigm that consists of creating, managing, supporting, and reusing. The creation process consists of developing applications by reusing and assembling components. The management process involves managing the selection of components according to requirements and costs. The supporting process carries out maintenance activities of the developed applications and delivers existing components from repositories. The reusing process consists of collecting and analyzing requirements in order to select the fitting components.

The main objectives of CBSD are to shorten the software development life cycle by delivering and deploying software applications more quickly, increasing reliability, and reducing the development effort and cost by integrating commercial off-the-shelf components (COTS) and newly developed components to construct large and complex systems. Heineman and Council summarize three main goals of CBSE (Heineman & Councill, 2001):

- Supporting the development of components as reusable entities. In this context, the benefits of reusing are productivity gains, quality gains (software systems are more dependable), and time-to-market gains.

- Supporting the development of software systems as assemblies of components through defining, implementing, and composing new and third-party components.
- Facilitating maintenance activities by replacing, updating, upgrading, and customizing the concerned components.

Although the benefits yielded by adopting CBSD, several issues should be addressed before building component-based applications:

- Trustworthiness issues: components come from multiple and independent sources before they are integrated into larger systems. The absence of countermeasures can conduct to use malicious or not well-tested components within these systems, which may break down the whole system. Therefore, the solidity of a system is correlated directly with the reliability of its components.
- Reusability issues: improved reusability implies adding complexity to the component since improving generality requires adding more generic interfaces and operations, which may decrease understandability and usability (Sommerville, 2011).
- Integration and interaction issues: components with different complexity levels, various configurations, and incompatible interfaces may affect the interaction between components, the integration process, and the overall quality of the component-based system. Additionally, as the number of components increases, the complexity and the number of interactions between components increase as well.
- Testing issues: testing difficulties at the unit level, especially with black-box components where the source code is unavailable are addressed in the literature (Tiwari & Kumar, 2020). Besides, integration testing of components imported from different repositories is more difficult than traditional software integration testing (C. Szyperski et al., 2002).
- Quality issues: the above-mentioned issues can affect negatively the quality of the developed components, and thus the overall system quality.

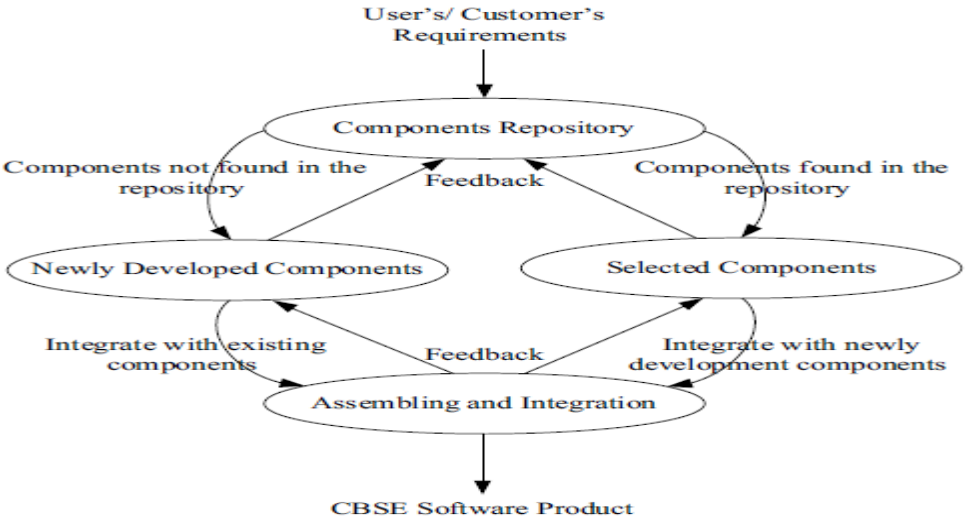

Figure 2.1. Concurrent CBSD process (Tiwari & Kumar, 2020)

2.2 Characteristics of CBSE

CBSE is a sub-discipline of software engineering characterized by the following main properties:

- **Reusability**: reusability is the pivotal and the backbone property of CBSE. Krueger defines it as "the process of creating software systems from existing software rather than building software systems from scratch" (Krueger, 1992). Johnson and Harris define it as "the process of integrating predefined specifications, design architectures, tested code, or test plans with the proposed software" (Johnson & Harris, 1991).
- **Composability**: composability is one of the primary properties of CBSE, in which components are reusable and composable software entities. Therefore, an application is constructed by composing different independent components. For a component to be composable, it should be designed in such a way that all external interactions should be performed through publicly well-defined interfaces.
- **Maintainability**: "maintainability represents the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements" (*ISO 25010*, n.d.). In the context of CBSE, maintenance is much easier than monolithic applications because systems are made of reusable and composable components. In this regard, components can be added, updated, removed, or replaced according to the software requirements (Tiwari & Kumar, 2020).
- **Shorter development cycle**: the CBSD paradigm consists of decomposing complex applications into smaller and manageable modules. Therefore, instead of starting the development cycle by coding modules from the first line of code, components are selected, imported, and composed according to the requirements of modules under development. This process can decrease the development time significantly. In addition, modules can be developed concurrently, which makes the development cycle shorter.
- **Improved quality**: generally, the CBSD paradigm permits the integration of pre-tested components at least at the unit level. Subsequently, other tests are performed, such as integration and system tests. These tests contribute to improving quality and constructing more robust and consistent application. Additionally, CBSD is intended to promote maintainability, reusability, flexibility, and extendibility.
- **Adaptability**: "Adaptability is the degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments" (*ISO 25010*, n.d.). A software system with high adaptability is easy to adapt to the changes that occur to its architecture and environment. In traditional software development, components are designed for specific purposes, which creates tight-coupled relationships with other components. As a result, updating a component may propagate changes to other components, which complicates

the updating process and decrease the adaptability of the overall system. Contrarily, in CBSD this process is more flexible, in which components can be added, updated, or replaced easily without affecting or modifying other components.

2.3 Component-based software life cycle

A software development life cycle (SDLC) or a software process is a methodology or a detailed plan that defines a set of related activities that lead to producing a software system. These activities include designing, developing, testing, and maintaining a software system before its disposal. The development process can start from scratch or by reusing and integrating pre-constructed components. Many development paradigms are proposed to makes the development process more disciplined, systematic, and easy to understand and follow by stakeholders. Despite the adopted paradigm, quality assessment and control can be performed in conjunction with any paradigm throughout its development stages (early, medium, and late assessment). However, the application and integration of the evaluation process may differ from one paradigm to another. In this section, we will outline both traditional and advanced paradigms before presenting the component-based software paradigm. Furthermore, we will discuss briefly the possible stages in which we can integrate MS-QuAAF into these paradigms.

2.3.1 Traditional software engineering paradigms

Traditional software engineering paradigms can be categorized into three broad categories (Tiwari & Kumar, 2020): classical paradigms, incremental paradigms, and evolutionary paradigms. In this sub-section, we present one paradigm for each category.

2.3.1.1 The waterfall paradigm

The basic waterfall paradigm is an organized, systematic, and sequential development process, in which the succeeding activity should not start until the previous one has been accomplished. As a result, the basic waterfall model cannot adapt to enhancements and requirements change during development. For these reasons, Winston Royce made some changes that allow feedback from one activity to another, which made the development process not simply linear (Royce, 1987). Therefore, the documents produced at each phase are modified to reflect the change made, which permit to fix problems that emerged at each development phase (figure 2.2). The main stages of the waterfall paradigm can be described as follows (Sommerville, 2011).

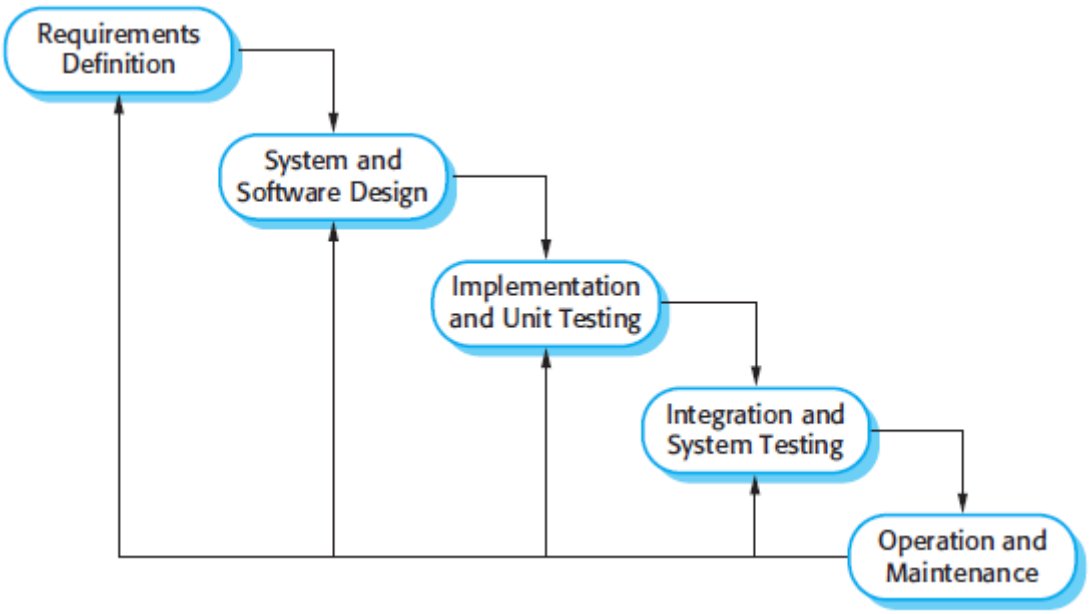

Figure 2.2. The Waterfall model

- *Requirement analysis and specification*: this activity describes what a system is supposed to do by specifying its functionalities and constraints on its operations according to the system's stakeholders.
- *Software design*: the overall design followed by detailed design are performed to construct the software architecture that describes system abstractions and the relationships among them. At this stage, quality assessment consists of evaluating the designed architecture and its conformance to the specified requirements.
- *Implementations and unit testing*: the established design is implemented as a set of executable program units. Subsequently, unit testing is performed to check that each unit meets its specification.
- *Integration and system testing*: the developed units are integrated to construct the complete system, which is verified and validated to ensure the satisfaction of stakeholders' requirements. Normally, the accomplishment of this phase means that the software is ready to be delivered to the end-user.
- *Operation and maintenance*: in this phase, the system is already operational. Maintenance activities involve correcting bugs that are not discovered in earlier phases, improving implementation, enhancing the system, and supporting new requirements.

Integrating quality assessment into the waterfall paradigm can be performed possibly at four main stages: design, implementation, integration, and maintenance. At the design stage, the early assessment consists of evaluating the designed architecture and its conformance to specifications. At the implementation and integration stages, the architecture is evaluated against the models defined at the design stage. At the maintenance stage, the architecture must be evaluated at each time new features and functionalities are added, updated, or removed.

The waterfall paradigm is an organized, manageable, and easy to implement process as long as the requirements are very clear from the beginning and they are not expected to change during the development process. However, it is very rare to start a project where all requirements are available at the earliest development stages. Additionally, its sequential nature means that concurrent phases are not supported, which implies that a development team cannot start a phase until the previous one has been finished (blocking states (Bradac et al., 1994)).

2.3.1.2 The incremental development paradigm

The incremental development model follows the sequential paradigm of the waterfall model in an iterative manner, in which the system is built incrementally. Each iteration produces a system release (increment) by passing through requirements specification, designing, coding, and testing activities. The deliverable of the first increment represents the core product where basic and highly prioritized requirements are tackled. In the next increment, more requirements and functionalities are added to the previous release. This process is repeated iteratively until all requirements of the desired system have been satisfied.

In our vision, assessing a software system that adopts the incremental development model must be performed at the increment level. At each increment, the designed and implemented architecture should be assessed before passing to the next increment. Consequently, the quality of the software can be controlled and improved throughout the development process incrementally.

The incremental paradigm has many advantages comparing to the classical waterfall paradigm. First, the development process can start with the most prioritized requirements without the need to collect all requirements in advance, which allows producing operational software with basic features more rapidly. Second, since the size of each iteration deliverable is small, the cost of requirement changing is reduced and the testing and debugging are much easier. Third, this paradigm is more flexible since new customer requirements can be added easily in the subsequent releases.

Although these advantages, many problems may emerge by adopting this paradigm. First, incorporating or changing requirements each time may lead to new and unforeseen errors, which can be costly in terms of time and money (Tiwari & Kumar, 2020). Second, adding increments may cause quality degradation and architecture deviation from the designed one unless money and time are spent on architecture improvement (Sommerville, 2011).

2.3.1.3 The spiral development paradigm

The spiral paradigm is a risk-driven development process and one of the most important SDLC. The development process is represented as a spiral rather than a linear set of activities with backtracking and feedbacks across these activities. Each loop of the spiral represents a development phase. More specifically, a loop is divided into four quadrants. The first quadrant specifies the objectives of the current phase, defines solutions, and identify risks. The second

quadrant concerns risk assessment and reduction. The third quadrant tackles development, construction, and verification activities. The last quadrant involves planning activities for the next loop.

The spiral paradigm is an interactive process, in which clients are involved in each development phase. It provides flexibility when changing/adding new requirements by accommodating them in the next loop. Additionally, the spiral model is especially used for risk recognition, assessment, and diminution at each development cycle. For this reason, the spiral model is the only one that integrate automatically the quality assessment in its development paradigm.

Although these advantages, the spiral paradigm is complex and not suitable for small projects because it is costly in terms of money and necessitates more time to plan. Besides, it requires risk experts to manage and mitigate risks, otherwise, risks are not assessed and managed properly and the project may go to fail.

2.3.2 Advanced software engineering paradigms

Over the years, numerous researchers, experts, and enterprises are involved in improving the software development process to overcome traditional paradigms' liabilities. This led to develop and evolve a set of advanced software engineering paradigms, such as clean room, agile, and CBSD. These paradigms changed imperatively the way of evaluating software architecture. For Example, some paradigms may allow promoting continual architecture evaluation; others may allow reducing the evaluation effort.

2.3.2.1 The Agile development paradigm

The Agile SDLC is a combination of incremental and iterative development processes with a focus on customer satisfaction. It includes a set of guidelines and practices performed by collaboration between self-organizing teams and their customers. The Agile paradigm breaks the development process into small increments, in which each iteration lasts from one week to three or four weeks. Each iteration includes a set of activities performed simultaneously by cross-functional teams. These development activities involve requirements gathering, design, development, testing, deployment, and feedback. At the end of each iteration, a new release of the working system is delivered to customers, which permits adapting quickly to changes. In this regard, the essence of the Agile paradigm is to promote continual improvements, iterative design, customer collaboration, and adaptiveness and rapid response to changes. The architecture evaluation process can be integrated into each iteration concurrently with other activities, in which designed and implemented architecture are assessed. This allows discovering and fixing design flaws and errors earlier at each increment release, which allows improving the overall quality of the final product and thus reducing maintenance effort and cost.

Contrarily to other paradigms, teams consist of all stakeholders, in which customers are considered as an integral part of the development team where the customer's voice is replaced

with the customer's story (Tiwari & Kumar, 2020). Changes are always welcomed, which allows adaptation at every stage of the development process. Time and overall cost are reduced because many activities are performed in parallel and errors can be caught and fixed quickly. Rollback decision can be taken at any stage and only the concerned increment can be affected.

On the other hand, adopting the agile model can engender many problems and shortcomings. The development process can become unmanageable when frequent changes affect increments because requirements are not clear or the customer is not satisfied. The Agile model depends heavily on customers' collaboration, and thus a shortage of documentation can be addressed. In this context, the lack of documentation may cause difficulties when maintaining the system or transferring information and technologies to new stakeholders. In addition, the Agile is not the right paradigm for small or low-budget projects.

2.3.2.2 Component-based software paradigm

The main idea of the component-based software paradigm is reusing pre-existing components to construct software systems. Embracing this development principle has several consequences for the software lifecycle (I. Crnkovic et al., 2006), in which we distinguish two separate CBSD processes:

- *Development for reuse:* this process concentrates on developing components that will be reused in other systems.
- *Development with reuse:* this process is concerned with developing new systems using pre-existing services and components.

When starting the development of a new system, we do not know what components are available; therefore, a new separate process of discovering and validating components will appear. Generally, these two processes constitute the main development activities for many component-based software paradigms. Figure 2.3 depicts these activities as follows.

1. *Requirements analysis and specification*: this activity consists of analyzing requirements gathered from the customer to remove deficiencies and ambiguity. The goal is to analyze the likelihood solution designated to meet the customer's requirements. In the CBSD, specifying requirements should take into account the availability of components that satisfy these requirements. If possible, the whole system will be constructed from pre-existing components; otherwise, unavailable components should be implemented. To squeeze all advantages of the CBS approach, requirements can be negotiated and modified to be fulfilled with the pre-existing components (I. Crnkovic et al., 2006).
2. *System design*: a complete architecture of the system should be defined at this stage, in which components and their relationships are identified based on the customer's requirements. However, this stage is tightly related to the availability of components as well as the requirements stage. Therefore, the availability of components and their

models (EJB, CORBA…etc) are the main determiners of the system design. For instance, using a component model that requires a specific style, such as client-server will oblige the system design to follow that style.

3. *Components identification*: this stage is unique to CBSD, in which three sub-activities are defined (Sommerville, 2011):
    - *Component search:* this activity consists of seeking candidate components that should meet the system requirements. In this context, components can be searched locally in the company repositories or from external trusted vendors.
    - *Component selection:* the selection of suitable components starts once the candidate components are identified. In an ideal case, candidate components match perfectly the requirements. In other cases, the selection is much more complex. The mapping between candidate components and the system requirements are not clear. Therefore, architects and developers must find the best composition that can cover these requirements.
    - *Components validation:* once components are selected; they must be validated by checking if they act as was expected. The test effort is correlated directly with the trustworthiness of the selected components. If components are imported from trusted suppliers where components are pre-tested, then testing may not be required. On the other hand, if components are developed by unknown sources, then test cases must be performed before integrating the component into the system.
4. *System integration:* this step consists of integrating components to build the executable system architecture. The integration consists in its turn of downloading components from repositories, registering, and deploying them into the component container.
5. *System test/ validation:* similar to traditional development models, tests and verifications are performed to check that the overall system meets the specified requirements. Assembling components may generate errors that require checking contractual interfaces (input and output) of the involved components. However, locating errors can be difficult especially when using black-box components brought from different suppliers (I. Crnkovic et al., 2006).
6. *Maintenance:* this activity encompasses some steps that are similar to the steps of the integration activity because new or modified components are integrated into the system. However, changing the glue code is probably required to support the new modifications.

Moreover, the system must be tested to fix incompatibility issues or broken dependencies.

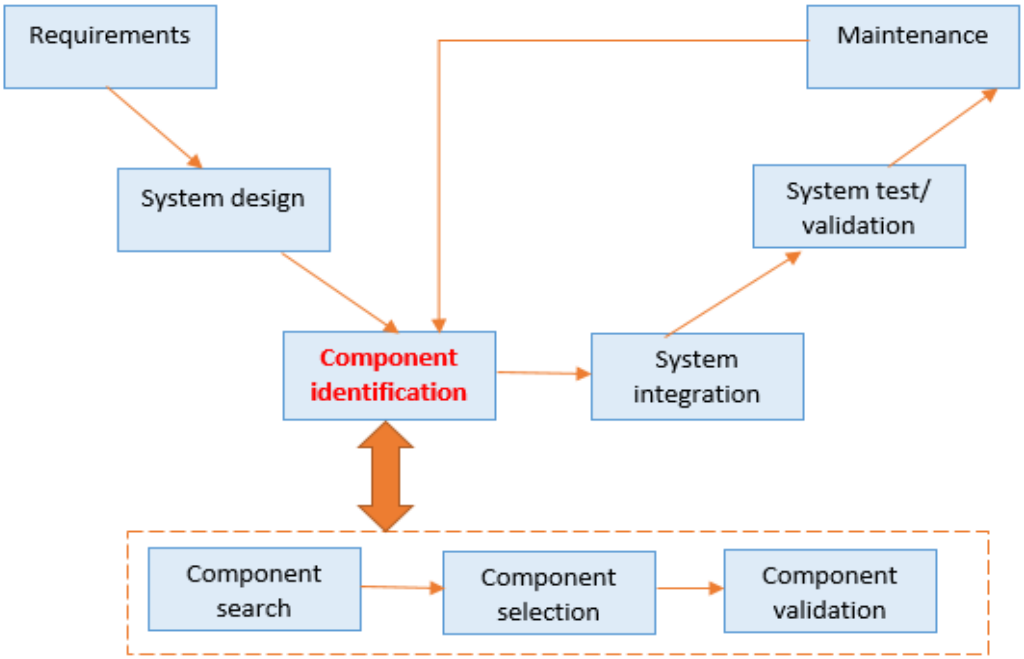

Figure 2.3. A V development model adapted to CBS paradigm

In this dissertation, the defined quality assessment approach can be integrated into the development for reuse process and development with reuse process. The former integration allows separate evaluation of components' internal architecture; however, this is possible only with white-box components. The latter promotes architecture evaluation at three development stages, which are system design, system validation, and maintenance. The evaluation at the design stage permits the assessment of the complete architecture resulted from interweaving components in order to detect and fix potential design flaws before implementation. At the validation stage, the implemented architecture is evaluated according to the designed one, and against the CBSD's properties, such as reusability and composability (section 2.1). At the maintenance stage, the architecture is evaluated when new components are added, updated, or removed.

One of the particularities of component-based systems is that evaluation effort is reduced significantly because most of the components (generally imported from trusted suppliers) are pre-tested and evaluated. Consequently, the evaluation may concern only the overall architecture constructed by combining these components.

2.4 Software architecture and components

2.4.1 Architecture

Software systems are built to satisfy organizations' business goals. The bridge between these goals and the final concrete system is called architecture. There is no universal definition of

software architecture; however, we attempt to provide the most known definitions in the literature.

Definition 1. The software architecture is the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both (Bass et al., 2012).

Definition 2. It can be defined as a set of significant decisions about the organization of a software system. These decisions include the selection of structural elements and their interfaces required to compose the system, the collaboration among those elements (the behavior), the composition of these elements into large subsystems, and the architectural styles that guide this organization (Rumbaugh et al., 1999).

Definition 3. "Software architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution" ("IEEE Recommended Practice for Architectural Description for Software-Intensive Systems," 2000).

Definition 4. "The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time" (Garlan & Perry, 1995).

Definition 5. "A software architecture is a set of architectural elements that have a particular form. Three different classes of architectural elements are distinguished" (Perry & Wolf, 1992b):

- The processing elements: represent components that transform data elements.
- The data elements: represent elements that contain data that is transformed by the processing elements.
- The connecting elements: represent the glue that interrelates different components to each other.

All these definitions state that architecture is a set of structural elements held by relationships. These elements should support reasoning about the system and its attributes that include functional and non-functional requirements considered important to some stakeholders. Therefore, architecture is an abstraction of the system because it shows only information of interest, such as elements and their relationships, and omits all the unnecessary details.

2.4.2 Components

In the CBSE community, components represent independent and reusable building blocks that can be composed with other components through predefined and opened interfaces to construct a software system. In the literature, various academic definitions describe components in various ways. Some of these definitions are provided below.

Definition 1. "A software component is a unit of composition with contractually specified interfaces and context dependencies only. A software component can be deployed separately and is subject to composition by third parties" (Clemens Szyperski & Pfister, 1996).

Definition 2. "A component is a physical, replaceable part of a system that packages implementation and provides the realization of a set of interfaces" (*About the Unified Modeling Language Specification Version 1.3*, n.d.).

Definition 3. "A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard." (Heineman & Councill, 2001).

Definition 4. "A software component is an architectural entity that (1) encapsulates a subset of the system's functionality and/or data, (2) restricts access to that subset via an explicitly defined interface, and (3) has explicitly defined dependencies on its required execution context" (Richard N. Taylor et al., 1996).

The above definitions agree that a component is an independent, replaceable, and modifiable unit that can be deployed and composed with other components according to composition standards and through publicly defined interfaces. In the context of CBSE, Sommerville defined five essential characteristics of a software component as follows (Sommerville, 2011).

- *Standardized*. A component must adhere to a standard component model. A model should define interfaces, documentation, deployment, and usage information.
- *Independent*. A component should be able to be composed and deployed without needing to use other specific components. If a component wants to use external services, then they should be specified as *required* in the interface specification.
- *Composable*. A component should provide a set of publicly defined interfaces to be composable and able to interact externally with other components.
- *Deployable*. A component must be self-contained and able to operate as a stand-alone entity on the target component platform. Additionally, components are usually delivered as binary entities that do not need to be compiled before their deployment.
- *Documented*. A component must be fully documented to allow users to decide whether to use the component that meets their needs or not.

2.4.2.1 Basic elements of components

2.4.2.1.1 Interfaces

An interface of a software component can be defined as a specification of its access point and the medium by which components connect (Ivica Crnkovic & Larsson, 2002; C. Szyperski et al., 2002). An interface can define a set of named operations that can be invoked by the component's client. If a component can provide multiple services, then multiple access points are defined through multiple interfaces. Two types of related interfaces reflect the services

provided by the component and the services required for the component to operate correctly (figure 2.4):

- Provided interface: this type of interface exposes the services provided by the component to other components. It defines a set of operations (methods) that can be called by the users of the component.
- Required interface: this type of interface specifies the services that must be provided to the component by other components to operate correctly. Thus, the absence of these services makes the component nonfunctional and inoperative.

2.4.2.1.2 Implementation

Contrarily to interfaces, implementation refers to the internal and concrete definition of the component, which is the source code. The implementation is hidden in some components and visible in others. According to the implementation visibility, we can distinguish two mains types of components:

- Black-box components: the implementation is decoupled and hidden behind interfaces. Therefore, the component's clients know no details beyond the interfaces and their specifications. The architecture can be built according to these specifications since knowledge about the internal implementation is not required in this type of component.
- White-box components: In the white-box components, the implementation is available, which may enhance the understandability and thus allows customizing the component to fit with the client's needs.

2.4.2.1.3 Specification

In the CBSE, we can distinguish two types of component specification, a syntactical specification, and a behavioral specification.

- Syntactical specification: this form of specification is concerned with describing interfaces, in which the signature part that represents named operations and their input/ output parameters are described. There are many techniques used to specify component's interfaces of different technologies, such as IDL (Interface Description Language) for CORBA (Common Object Request Broker Architecture) and COM technologies, and the Java programming languages for the Sun's JavaBeans (Ivica Crnkovic & Larsson, 2002). However, syntactical specification is not addressed to specify the overall behavior of components. As a result, *contracts* are proposed by Meyer (Meyer, 1992) as an extension to syntactical specification by adding the notion of behavior.
- Contract (behavioral specification): a contract specifies for each operation within the component the constraints that the component should maintain. These constraints represent the pre and post-conditions that constitute the specification of the component's behavior. Moreover, contracts specify the interaction between

components in terms of the set of involved components, the role of each component, and the invariant to be maintained by each component.

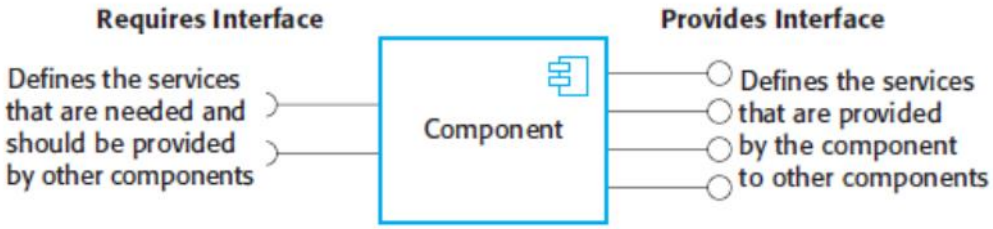

Figure 2.4. Component's interfaces (Sommerville, 2011)

2.4.3 Connectors

Software components are meant to provide functionalities supposed to constitute the application business logic, whereas connectors are designated to manage the connectivity between these components. This organization permits to keep the component's focus on application and domain-specific concerns by separating the computation concerns handled by components from interaction concerns handled by connectors (R. N. Taylor et al., 2009). A software connector can deliver one or more services, such as transferring control and data among components, messaging, transaction, etc. Taylor defines four main classes of interaction services that a connector can provide (R. N. Taylor et al., 2009):

- *Communication*. Many communication services that consist of transferring data between building blocks are provided by connectors. In this context, components exchange frequently messages, data to be processed, and computation results by the means of connectors.
- *Coordination*. Connectors provide coordination services by supporting control transfer between components. Method invocations, function calls, and load balancing management are examples of coordination connectors.
- *Conversion*. Managing the interaction among heterogeneous components is not a trivial task. Conversion connectors allow overcoming interaction issues between these components by providing conversion services, such as wrappers and data formats converters.
- *Facilitation*. This service is dedicated to facilitating and optimizing the interaction between components. For instance, concurrency control, load balancing, and scheduling services are provided by connectors to meet certain non-functional requirements and reducing interdependencies between involved components.

Based on this categorization of interaction services, Taylor et al. defined eight different types of connectors, which are procedure call, event, data access, linkage, stream, arbitrator, adaptor, and distributor.

Although software connectors have not been addressed and studied to the same extent as components, they have an important role in streamlining interaction among components that expose heterogeneous functionalities developed by various organizations at different times and locations.

2.4.4 Architecture modeling

Models represent an abstraction of a system, intended for understanding, designing, or explaining aspects of interest of that system (Dori, 2011). They are used to support many activities throughout the development process, such as design, analysis, and verification and validation (V&V). In this connection, Taylor et al. defined an architecture model as an artifact that aimed at capturing some or all design decisions that comprise the system's architecture using architectural modeling notations (R. N. Taylor et al., 2009). These notations are nothing but languages and means of capturing architectural decisions. Architecture modeling notations range from informal and rich to highly formal and narrow. The available notations can be divided into three main categories as follows.

- *Informal notation.* The informal notation is not designed to be analyzed and processed by machines, in fact, it is dedicated to being checked and understood by humans. However, informal representation is ambiguous and confusing because it gives inconsistent interpretations and it cannot be analyzed mechanically (Jackson, 2019). Graphical diagrams that consist of boxes and lines or natural languages are good examples of informal notations. Usually, this type of notation is intended for non-technical stakeholders, such as managers and customers.
- *Semi-formal notation.* The semi-formal notation is widely used in architecture modeling. It is intended to be useful by both technical and non-technical stakeholders. The essence of semi-formal languages is to try to make a balance between a high degree of precision and formality on the one hand, and expressiveness and understandability on the other hand (R. N. Taylor et al., 2009). Models are expressed using standardized notation such as graphical elements and constructions rules; however, it does not provide a complete treatment of the meaning of these elements (Bass et al., 2012). This type of notation can be analyzed manually by humans or automatically by machines. A broadly and most popular used notation is the Unified Modeling Language (UML) (*Welcome To UML Web Site!*, n.d.). UML is syntactically rich and fundamentally based on graphical notation that uses textually annotated graphical symbols. However, the ambiguity about the meaning of these symbols can produce different interpretations of the same UML diagram. Generally, semi-formal representations may suffer from precision issues, which make performing sophisticated analysis hard to achieve.
- *Formal notation.* The formal notations define and describe structure and behavior precisely and objectively (Jackson, 2019). Contrarily to semi-formal notation that provides only formally defined syntax, formal representation provides also formally

defined semantics. Besides, formal notations are amenable to automatic analysis, such as Alloy (Jackson, 2012), the Z notation (Spivey & Abrial, 1992), and Wright (R. Allen & Garlan, 1997), which make this type of notation intended only for technical stakeholders.

2.4.5 Architecture Description Languages (ADLs)

As architecture became a dominating substance in developing software systems, Architecture Description Languages (ADL) have emerged as formal notation that can be used to represent that architecture in an unambiguous way (Clements, 1996a). More specifically, an ADL is a notation used to express architecture models in order to capture and document the architectural decisions taken to build a system's architecture. Medvidovic *et al.* claim that an ADL must explicitly model *components*, *connectors*, and their *configurations*. In addition, it must provide the *tool support* to be considered useful and usable. These four elements should be the fundamental constituents of any proposed ADL (Medvidovic & Taylor, 2000). Clements defined five requirements for a language to be an ADL as follows (Clements, 1996b).

- An ADL must support the creation, refinement, and validation of an architecture.
- An ADL must be able to represent most of the common architectural styles.
- An ADL must be able to express architectural information through system views.
- If an ADL can express the implementation level, then it must have the ability to match more than one implementation to the architecture views.
- An ADL must support analytical capabilities based on the architecture level, or the capabilities to generate prototype implementations.

However, there is always a continuous debate about which notation can or cannot be called an ADL? We agree with Taylor et al. by considering any notation used to capture and document the main architectural decisions as an architecture description language.

There was a myriad of proposed ADLs that can be classified as general-purpose languages or domain-specific languages. We will discuss briefly a small set of these ADLs as follows.

- **Darwin:** Darwin is a general-purpose ADL used for specifying structures of component-based systems. It provides graphical representation along with well-defined textual syntax to describe components and their interconnections in order to capture architectural decisions more rigorously and formally. It should be noted that Darwin does not support connectors explicitly; however, components that play the role of connectors by facilitating interconnection can be interpreted as connectors. Besides, Darwin is designed for structural modeling and not for describing other architecture's aspects, such as behavior (Magee et al., 1995).
- **Rapide:** Rapide is an event-based language designed for specifying architectures of distributed concurrent systems. An event is an object generated by calling an action, such as a procedure's call or reply. Events are organized into partially ordered sets and

related by causal and timing relationships. Rapide allows behavioral analysis and simulation of architectures at the early stages of the development process. It provides architecture constructs for defining executable prototypes before the implementation stage, in which synchronization, concurrency, timing, and dataflow are represented explicitly (Luckham et al., 1995). However, Rapide is focused on event-based modeling to addressing the dynamic aspects of architecture at the expense of other aspects. Additionally, its notation is obscure and difficult to learn.

- **Wright:** Wright is an ADL developed to describe and analyze software architecture and architectural styles using concepts such as *components*, *connectors*, *ports* and *roles* (interfaces), *attachments*, and *styles*. It is based on the formal specification of the abstract behavior of components and connectors. Wright uses CSP-like (Communicating Sequential Process) notation to describe this behavior, predicates to characterize styles, and static checks to determine the consistency of architectural specifications (R. J. Allen, 1997). Similar to Darwin and Rapide, Wright lack support to refine architectural specification into implementations.
- **Acme:** Acme is an extensible ADL developed to express the structural design decisions using seven basic constructs: component, connector, port, role, system (configuration), attachment, and representation. Originally, Acme was designed to provide an interchangeable language for development tools rather than being an ADL (Garlan et al., 2010). The language should integrate ADL tools by providing an interchange format for exchanging architecture information. However, this strategy was never applied at a wide scale, probably because of the semantic differences among the target ADLs (R. N. Taylor et al., 2009).

Additionally, other languages that can be classified as ADLs can be found in (Clements, 1996b; Medvidovic & Taylor, 2000; R. N. Taylor et al., 2009).

2.5 Architecture analysis

Architecture analysis is the activity that allows discovering important system properties using architectural models. In this context, the analysis can reveal inappropriate design decisions and subtle flaws at early stages, getting early answers about the system's complexity and size, checking the conformance of the architectural models to constraints and design guidelines, and the satisfaction of functional and non-functional requirements. Taylor et .al defined four main goals of architecture analysis (R. N. Taylor et al., 2009):

a. ***Completeness*.** This goal can be divided into external and internal analysis goals. Assessing the architecture's external completeness consists of verifying whether the architecture captures adequately all the system's fundamental functional and non-functional requirements. However, this can be hard to achieve due to architecture's high complexity and size. Assessing internal completeness consists of verifying whether architecture's elements have been fully captured according to the architectural decisions taken and the

modeling notations used to model such architecture. Internal completeness is easier to assess than the external one because it is amenable to automation.

b. ***Consistency.*** Consistency is an internal property that allows ensuring that different architectural elements, rules, and constraints do not contradict one another. Architecture models can help in discovering inconsistencies resulted from architectural decisions captured by these models. For instance, the following inconsistencies can be detected at the model level:
   - **Name inconsistencies** can occur when components, connectors, and provided services may have similar names. At the architectural level, this may engender a problem, especially with large architectures. In this context, determining which components are wrongly accessed is difficult.
   - **Interface inconsistencies** may occur when a component's required interface and another component's provided interface have the same name, nonetheless, their return types and parameter lists (signatures) may differ. Moreover, interface inconsistencies incorporate name inconsistencies and not the opposite.
   - **Behavioral inconsistencies** occur when the names and interfaces of components that request and provide services match, but their behaviors do not.
   - **Interaction inconsistencies** are the result of violating interaction protocols between components even their names, interfaces, and behaviors are consistent.
c. ***Compatibility.*** Compatibility is an external property intended to ensure the adherence between architectural models and the specified architectural styles, tactics, constraints, and design guidelines. The analysis for compatibility can be straightforward and amenable to automation if these design rules are specified formally; however, the analysis would be challenging with semi-formal and informal specification.
d. ***Correctness.*** Correctness is an external property that determines whether architecture is correct according to the external system specification. The correctness can be achieved if and only if the architectural decisions made realize entirely those specifications. Besides, the implemented architecture is considered correct according to the designed architecture if and only if the implementation captures and realize all the key architectural decisions of the designed architecture.

Additionally, architecture analysis permits to discover architecture degradation that comprises two related architecture mismatches, which are architecture *drift* and *erosion* (Perry & Wolf, 1992a):

- *Erosion*. Architecture erosion is the result of modifying the system by introducing new architectural decisions that violate and disregard its fundamental rules and prescriptive architecture (De Silva & Balasubramaniam, 2012). The accumulation of these modifications over time will create a mismatch between the descriptive (realized or implemented architecture) and the prescriptive architecture (intended architecture). In

other words, the implemented architecture will diverge from the intended architecture. Architecture erosion may lead to disastrous results and system failure.

- *Drift*. Architecture drift is the result of introducing new modifications into the descriptive architecture that are not encompassed by the intended architecture. These changes do not violate necessarily the prescriptive architecture's architectural decisions; however, they may lead to future inadaptability and a lack of clarity and coherence that make the architecture more obscure and easy to violate (Perry & Wolf, 1992a; R. N. Taylor et al., 2009).

2.5.1 Architectural characteristics to be analyzed

Architectural models permit to address many architectural concerns of given software architecture. These concerns can be structural, dynamic, or non-functional.

a. ***Structural characteristics***. These characteristics denote the organization of the architectural elements that constitute the software architecture. Therefore, the analysis should focus, for example, on connectivity and containment relationships among components. In other words, determining whether the architecture is well-formed and structured by verifying the adherence to the specified architectural styles, constraints, and tactics. For instance, verifying whether the connection between two components is authorized, checking disconnected components and subsystems from the rest of the architecture, determining the conformance to a specific style, or ensuring that the number of provided interfaces of a specific component should not exceed five.
b. ***Behavioral characteristics***. Well-formed architecture is of great importance to building software systems of quality; however, it can be of limited utility if the system does not provide the expected behavior specified at early stages. The analysis must consider two main types of behavior. The former is the internal behavior of individual components; however, the analysis can be restricted to the public interfaces if the component's internal implementation is hidden. The latter is the composite behavior or the system-level behavior resulted from combining components.
c. ***Non-Functional characteristics.*** Non-Functional characteristics define how the system is supposed to be, such as modifiability, availability, and performance. They constitute restrictions or constraints on the design of the system across its different components and connectors. However, they are qualitative in nature and often miscomprehended, which makes them difficult to analyze and assess.

2.5.2 Analysis techniques

Various techniques of architecture analysis is available for software architects. These techniques can be categorized into three main categories (R. N. Taylor et al., 2009): inspection and reviews, model-based, and simulation-based analysis.

a. ***Inspection and reviews.*** This technique includes a set of activities conducted by different stakeholders (especially architects and developers) in which architectural

models are studied to ensure specific architecture properties, and then, to critique the architecture under analysis as a whole or some of its parts. Usually, inspection techniques are performed manually, which makes them tedious and expensive; however, they are advantageous when using informal notations to specify architecture. In addition, many task analyses that consider different architectural properties can be performed simultaneously by the relevant stakeholders.

b. ***Model-based analysis.*** This analysis technique relies entirely on the architectural description of the underlying system to explore architecture properties using dedicated analysis tools. The main goals of the model-based analysis are to verify consistency, internal and external completeness, and correctness. Some analysis tools that use this technique would generate automatically system implementation from architectural models, which may ensure the external completeness (adherence of the system's implementation to the architectural models) and correctness. Unlike inspections and reviews techniques, model-based techniques are less costly. However, models permit to analyze only explicit architecture properties and they cannot be used easily for implicit properties. The latter can be readily inferred by humans from existing information. Furthermore, model-based techniques cannot provide all answers to stakeholders, and they are coupled with other techniques from inspection and reviews and simulation-based analysis.

c. ***Simulation-based analysis*.** This technique necessitates generating executable and dynamic models of the system or its parts of interest, usually from source models. It should be noted that not all architectural models are amenable to simulation, such as informal models, because they lack the appropriate formalism (formal notation) required to enable their execution. The main goals of the simulation-based analysis are also completeness, compatibility, consistency, and correctness. However, this technique requires high technical expertise with simulation, architectural models, and modeling notations.

2.6 Conclusion

This chapter introduced generalities and main concepts of component-based software engineering and software architecture. These concepts are considered important and must be addressed before getting involved with architecture quality control and evolution, which is the central aim of this dissertation.

In a highly competitive and demanding software marketplace, CBSE has emerged as a new paradigm to develop software systems of quality more rapidly based fundamentally on reusability and composability principles. This paradigm has many advantages as well as some shortcomings; however, our main interest is the architecture quality of software systems constructed by this paradigm and not the paradigm itself. In this context, architecture specification and modeling using formal and semi-formal notations (section 2.4.4 and 2.4.5),

and model-based analysis (section 2.5.2) will be used intensively in the process of quality control.

Chapter 3

Software quality

In the previous chapter, we have presented some important concepts that concern component-based software and software architecture. The current chapter is complementary to the last one, in which we will discuss the concepts of quality in general and software quality in particular. The latter is the key element of the framework MS-QuAAF, and the heart and the soul of this dissertation.

For many decades, software quality is considered very critical and essential for organizations, and a key component for business success. Nowadays, the software marketplace is extremely competitive, which means that developing software systems of high quality is not only an obligatory but also a differentiator factor for companies (Kadri et al., 2021b). Critical sectors that depend heavily on software, such as homeland security and air control traffic expect to deploy error-free systems that are supposed to fulfill their quality requirements flawlessly. In this context, poor software quality is responsible for severe business and safety disasters (Carrozza et al., 2018). Therefore, companies must boost up software quality perception and integrate it profoundly into the development process.

This chapter is organized as follows. The first section depicts the concept of quality according to the perspective of well-known quality gurus, such as Shewhart and Deming, the comparison between those perspectives, and the presentation of our own perspective. The second section presents the main software quality management approaches, such as software quality assurance and software quality control. The third section explains software errors, defects, failures, and the difference between them. The fourth section presents the most well-known quality models, the comparison between them, and the relationship between these models and the framework MS-QuAAF.

Contents

3.1 What is software quality? 31
3.2 Software quality management 37
3.3 Error, fault (defect), failure 41
3.4 Quality models 44
3.5 Conclusion 53

3.1 What is software quality?

"*What is software quality?*" is a kind of question that generates different answers depending on whom we asking (consumers or producers of the system), requirements, type of software system, business and market environment, and so forth. In this section, we will attempt to

provide a set of definitions according to the point of view of some of the most-known quality pioneers, consumers, and producers. Furthermore, we will outline our vision about quality and quality control adopted in this dissertation.

Emphasizing these historical definitions as well as stakeholders' perspectives allow better understanding of quality, quality attributes, quality models, quality assurance, and quality control.

3.1.1 The quality according to Shewhart

Walter Shewhart was considered as one of the giants of the quality movement in the first half of the 20th century, and the founder of the Statistical Process Control (SPC) (Walter Andrew Shewhart, 1931). He developed a control chart, which is an essential tool used to control the process. The chart permits monitoring a process by providing upper and lower control limits. Therefore, a process is closely controlled if it performs inside these limits (O'Regan, 2014). He defined also a cycle model called the PDCA model (Plan-Do-Check-Act), which is a systematic approach to process control and problem-solving. In his influential book, he asserted that productivity and quality improve as process variability is reduced.

Shewhart argued that controlling the quality of a product must involve three steps: the specification of the standard of quality designed to meet the stakeholders' needs; the production of the desired product; and the verification of the conformance of the product to the specified quality standard (WALTER A. Shewhart, 1958). Obviously, the quality according to Shewhart is the conformance to requirements and quality standards. Thus, a piece of product is accepted or rejected according to the result of conformance checking.

3.1.2 The quality according to Deming

W. Edwards Deming, one of the major figures of the quality movement is an American statistician and business consultant influenced by Shewhart's work on SPC. He argued that improving quality might be achieved by reducing costs by decreasing rework of defective products, and thus improving productivity as reworking time decrease. Moreover, improving quality is 1) the responsibility and the efficiency of the top management that should adopt the appropriate management principles, and 2) it must be a constancy of purpose from all individuals to ensure success (O'Regan, 2014).

The quality according to Deming is defined by or measured by the satisfaction of customers. He defined good quality as *the predictable degree of uniformity and dependability with a quality standard suited to the customer* (Deming, 1986). In this context, a manager's quality definition is when specification and final product are met. On the other hand, a customer's definition is when the product meets his/her needs and expectations.

In his influential book *Out of the Crisis,* Deming defined 14 principles for total quality management to transform western organization style to customer-focused organizations:

1. Improving products and services by creating constancy purpose.
2. Adopting the new philosophy.
3. Building quality into the product and stopping depending on inspections because they are costly and unreliable.
4. Minimizing cost by working with a single supplier.
5. Improving constantly and forever the system and its processes.
6. Using institute training on the job.
7. Implementing leadership instead of supervising.
8. Eliminating fear.
9. Breaking down barriers between different staff and departments.
10. Eliminating Slogans
11. Eliminating numerical quotas.
12. Removing barriers to pride of workmanship.
13. Implementing education and self-improvement for everyone.
14. Accomplishing transformation is everyone's job.

3.1.3 Quality according to Juran

Joseph Juran, another giant in the quality movement defined quality as *fitness for use*. His top-down management approach is customer-based, in which customer satisfaction is realized when a product meets customer's needs (J. M. Juran & Godfrey, 1999). Accordingly, He claimed that quality issues are the direct responsibility of management that must ensure planning, controlling, and improving quality. These three activities are called *Juran Trilogy* described usually by a diagram with the cost of poor quality on the vertical axis and time on the horizontal axis (O'Regan, 2014). This trilogy is an improvement cycle intended for reducing the cost of poor quality as follows.

a. *Quality planning*. This activity consists of defining customers' needs, setting goals, defining requirements for the product, and developing plans along with stakeholders' expectations.
b. *Quality control*. This activity consists of evaluating performance, determining needs to be assessed, and taking actions and filling the gap between performance and goals.
c. *Quality improvement*. This activity consists of repairing defects and continually improving delivery and customer satisfaction.

Additionally, Juran defined ten steps to quality improvement as follows (J. M. Juran & Godfrey, 1999).

1. Creating awareness of the need and opportunity for improvement.
2. Setting goals for improvement.
3. Organizing to reach the goals.
4. Providing training throughout the organization.
5. Carrying out projects to solve problems.

6. Reporting progress (expected and achieved progress).
7. Giving recognition.
8. Communicating results.
9. Keeping scoring by sticking to the plan and tracking progress to reach goals.
10. Maintaining momentum by making annual improvement part of the regular processes of the company.

These steps have been evolved over the years to be suited to different segments and organizations.

3.1.4 Quality according to Crosby

Philip Crosby is a philosopher, consultant, and major figure in the quality movement. His two influential books *Quality is Free* (Crosby, 1980), and *Quality without Tears* (Crosby, 1995) outlined his perspective on quality. He defined *the zero-defect* (ZD) program that outlines his philosophy of *doing things right the first time.* His point of view on quality is clear-cut. The latter does not consist of some vague concepts, such as goodness, high or bad quality. Instead, it is unambiguously *conformance to requirements.* He argued that the term Acceptable Quality (or Defect) Level (AQL) is evidence of failure and a commitment to producing defective products. In this context, defects can be caused by two main reasons, a lack of knowledge or lack of attention of the individual (O'Regan, 2014). The former can be resolved by training. The latter is more difficult and requires changing the individual's mindset.

The outcome of the right implementation of the zero-defect program is improved productivity due to fewer defective products and reworking. The goal of this program is preventive in order to meet requirements the first time and every time. Poor quality is the responsibility of management. The latter should seed the program of zero defect in their employees' minds to achieve the desired quality and business goals. Crosby defined a fourteen-step improvement program to achieve this quality. The program necessitates management commitment and cross-functional and well-organized improvement teams to be successful. The state and the cost of quality should be determined, and corrective actions must be performed accordingly.

In summary, Crosby's philosophy consisted of three main principles:

1. Quality is conformance to requirements.
2. Prevention is the system of quality.
3. The only performance standard is the zero-defect program.

However, this philosophy does not take into account the difference in quality between products. For instance, a BMW car is of the same quality as a Dacia car if they are conform to requirements. Furthermore, the ZD program is not suitable for software systems where high complexity is often the source of defects rather than the mindset of software engineers.

3.1.5 Quality according to Ishikawa

Kaoru Ishikawa is a major figure in quality control techniques and training. His perspective on quality is customer satisfaction, in which managers should ensure to meet consumer needs. Although he believed in standards, he argued that consumer needs are the ultimate source of decisions making. He invented the fishbone diagram (cause and effect diagram) which is a pictorial diagram that depicts visually possible causes of a result. It is used for analyzing industrial processes to find the root causes of a specific problem. He proposed the *quality control circle*, which is a small group of employees who do similar work, and meet to analyze and identify problems, and recommending and implementing solutions in order to improve quality and productivity. In this context, Ishikawa recommended seven basic tools for quality control and improvement, which are the fishbone diagram, Pareto chart, process mapping and data gathering tools, graphical tools, run charts, scatter plots, and flowcharts (Ishikawa, 1985).

3.1.6 Quality according to IEEE Std.730-2014

According to IEEE 730-2014, quality is *the degree to which a software product meets established requirements; however, quality depends upon the degree to which established requirements accurately represent stakeholder needs, wants, and expectations* ("IEEE Standard for Software Quality Assurance Processes," 2014).

3.1.7 Comparison and contrast

The comparison and contrast between the above-mentioned quality icons allows us to extract many similarities as well as differences that concern their points of view, quality improvement processes, management responsibilities, performance indicators, and assessment tools.

Similarities: the first obvious similarity shared between them is putting the responsibility of product quality on the shoulders of the senior management. They argued that top management should plan, control, and improve quality by adopting the right principles. Furthermore, management has the whole responsibility of ensuring communication and coordination across the involved teams and the different levels of hierarchy by providing the necessary training and the appropriate work conditions.

The second similarity is that individuals involved in the production process must get training and continuous education to improve quality. According to Crosby, the lack of knowledge and training is one of the main reasons that cause defects. Therefore, an organization must set up training programs to level up the skills of management as well as their staff.

	Conformance to requirements	**Customer Satisfaction**	**Tools, approaches, and slogan**
Shewhart	X	X	Control chart, PDCA model
Deming		X	14 points, deadly and dreadful deceases

Juran		X	Juran Trilogy, 10 steps
Crosby	X		Zero defects, 14 steps
Ishikawa		X	Fishbone diagram, quality control cycle
IEEE 730-2014	X	X	/

Table 3.1. The comparison between the quality pioneers' viewpoints

Differences: By comparing the above quality definitions and perspectives, two main philosophies adopted by experts to define quality can be outlined (Table 3.1):

- *Conformance to requirements*: quality is defined as conformance to requirements. A product is judged of high quality if it meets the specified requirements. Crosby is the most commonly known for adopting such a philosophy.
- *Customer satisfaction*: quality is defined as customer satisfaction. A product is judged of high quality if it fulfills customer's needs and expectations. Deming, Juran, and Ishikawa are considered among the adopters of this philosophy.

The definition of Shewhart and IEEE 730-2014 standard is a combination of these two philosophies. They argued that a product should meet the specified requirements to generate customer satisfaction.

The second difference is about the performance standard. Shewhart defined the control chart to measure and monitor quality. A quality process is under control if it performs inside the lower control limit (LCL) and the upper control limit (UCL). Crosby proposed the zero defect, which is not just a motivational slogan, but a commitment and mindset to prevent defects. Although he did not mean that all outputs have to be perfect, he claimed that involved individuals should be committed to meet requirements the first time and every time. Juran did not define explicitly a performance slogan; however, he defined the *Breakthrough and Control* approach to achieve a new (higher) quality performance level (Joseph M. Juran, 1964). He described pictorially by a control chart the breakthrough, in which the old and the newly attained performance level are presented. On the other hand, Deming did not define any performance slogan. In fact, he argued that slogans cannot help anyone to do a better job.

The third difference concerns quality management and control. Crosby's approach to achieving quality is prevention rather than error detection and correction. He proposed the prevention process, which involves requirement establishment, measuring, comparing, and taking actions. The first three processes are defined to anticipating the possibility of errors occurring, whereas the last process is to take actions to keep errors from occurring. Additionally, He defined the fourteen steps program to achieve quality by performing the appropriate actions according to the state and the cost of quality. Deming's approach is also prevention to promote quality. The prevention is achieved by three processes, which are analysis, control, and improvement. He defined fourteen principles (obligations) to transform any organization no matter its size into a

customer-focused organization through quality. Juran's approach is to promote continuous improvement to achieve quality. He defined the "Juran Trilogy", which is an improvement cycle that involves planning, control, and improvement processes. Shewhart's systematic approach focuses also on continuous improvement. He proposed the PDCA cycle to control quality and solve emerging problems. Finally, Ishikawa's approach is based on inspection and detection to locate and solve problems. He defined the fishbone diagram for analyzing the industrial process in order to detect the main causes of specific problems. Furthermore, he proposed problem-solving techniques (section 3.2) to control and manage the quality improvement process.

3.1.8 Our perspective on software quality

The software quality viewpoint addressed in this dissertation is tight-coupled with Non-Functional Requirements (NFRs), which are characteristics of a software system called often quality attributes (QAs). The latter represent measurable (or testable) properties of a software system used to designate its goodness and how well it satisfies the needs of its stakeholders (Bass et al., 2012). Contrarily to functional requirements that refer to the purpose of the system or the intended functions supposed to be provided by the system, quality attributes permit to determine how the system is supposed to be. For instance, the system should be maintainable, secure, and available. Therefore, *maintainability*, *security*, and *availability* represent the system's quality attributes.

Accordingly, we can define the quality of a software system as the degree of achievement (satisfaction) of the specified quality attributes desired by all stakeholders and not just customers. For instance, architects and developers are interested in portability and modifiability, whereas customers are interested in efficiency and usability. However, QAs of a system do not have the same priority; therefore, a software is judged of high quality if it can meet tradeoffs among the competing NFRs. As a result, the quality of the software is correlated with the ability to promote and achieve competing quality attributes and manage the tradeoffs among these attributes.

This perspective on quality is the basis on which we developed the framework MS-QuAAF to control and assess the quality of software architecture. More specifically, the framework assesses the satisfaction of each QA separately, and then it calculates the overall satisfaction score.

3.2 Software quality management

Satisfying customer's needs, as well as the organization's objectives, necessitate the implementation of a successful quality strategy to achieve the desired quality level. A quality strategy consists of designing and constructing a solid quality management system (QMS) that is able of detecting defects at early stages in order to minimize cost and delivering high-quality outputs continually. A QMS is an aggregate of quality-related structures, plans, processes, tools,

policies, and infrastructures (Westfall, 2016). In this section, we will outline some quality management approaches and philosophies, such as total quality management, quality assurance, and quality control.

3.2.1 Software product stakeholders

Stakeholders represent individuals or groups who affect or affected by a software product and thus have a degree of influence over the requirements for that software (Westfall, 2016). They can be categorized into two main classes: *suppliers* (producers) and *acquires* (consumers) of the software.

Acquires of the software can be divided into two main categories. The first category is *customers* who select, request, and purchase the software in order to fulfill their business goals. The second category represents *users* who use actually the software directly or indirectly. In their turn, users can be divided into many groups that have different levels of skill, knowledge, roles, and privileges, such as novice users, occasional users, and advanced users.

Suppliers of the software include all individuals and groups involved with the development of the software system in its lifecycle, such as managers, architects, programmers, testers, maintainers, and evaluators.

The quality expectations on the consumer side are achieved when the software performs its functionalities as it is specified correctly and reliably. In other words, the software should meet the customer's desired functional and non-functional requirements. On the other hand, the quality expectations on the producer side are achieved by delivering software systems that conform to the specified requirements in order to fulfill contractual obligations. More specifically, the software should promote all the specified ***ilities,*** such as maintainability, reliability, functionality, and so forth.

3.2.2 Total Quality Management

Total Quality Management (TQM) is a management philosophy for customer-focused organizations. It focuses on quality continual improvement by developing a quality culture within the organization. In this regard, quality is a company-wide objective, in which total customer satisfaction is a first-class organization goal. TQM requires that all members of an organization participate in improving products, services, and processes. Furthermore, all functions in the organization must follow high standards, and all staff must be trained in quality management. More specifically, the TQM can be divided into four main parts (O'Regan, 2014):

- Customer-focus: customers are the determiner of the level of product quality. They have rights and expectations that need to be satisfied, which require the involvement and the training of all staff and the commitment of the senior management.

- Process: this involves the focus on the process and its improvement by using problem-solving to identify the root causes of problems, and thus taking the right decisions subsequently.
- Measurement and analysis: to promote continual improvement, analysis of the quality of the product and process must be performed repeatedly by setting up a measurement program with the organization's activities.
- Human factors: this involves integrating a quality discipline into the organization and developing a culture of customer satisfaction. This can be achieved via training and promoting effective communication to motivate employees to embrace the TQM philosophy.

3.2.3 Software Quality Assurance

According to the IEEE 730-2014 standard, Software Quality Assurance (SQA) is *a set of activities that define and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate for and produce software products of suitable quality for their intended purposes* ("IEEE Standard for Software Quality Assurance Processes," 2014). SQA is a preventive approach that involves all the planning and implementation activities that are integrated into all stages of the software development process starting from requirements definition until product release. These activities may include creating an SQA plan, setting checkpoints, applying software engineering techniques, defining testing strategy, controlling change, measuring change impact, and performing SQA audits. All these activities are performed to:

- Ensuring an acceptable level of confidence that the software system will conform to functional technical requirements, and thus achieving a suitable quality level.
- Ensuring an acceptable level of confidence that the software development process will conform to budgetary and scheduling requirements.
- Improving the efficiency of the software development process to boost up the achievement of managerial and functional requirements.

The quality in the organization is promoted by the quality assurance group. The latter is an independent group that provides an independent product assessment, which involves conducting audits to verify conformance, reporting the audit results to top management, and improving the software development process. Mainly, the essence of a quality assurance team is to act as the customer's voice to achieve the desired quality level and ensuring that quality is considered at each step in the software process.

3.2.4 Software quality control

Software Quality Control (SQC) is a set of activities designed to ensure that a developed product meets the functional and non-functional requirements. For this purpose, SQC can be performed with the help of some control techniques, such as inspection and testing to detect defects.

The inspection consists of a formal review of a deliverable to identify defects and provide confidence in its correctness. However, inspection is not meant for building quality into the product from the beginning because it is performed at the end of the production process. As a result, defective products are removed from the batch to be revised and reworked (O'Regan, 2014). On the other hand, software testing can be integrated into many stages of the development cycle to detect defects relatively earlier comparing to inspection techniques. Software testing consists of *white box* or *black box* techniques and may encompass unit, system, integration, acceptance, and performance testing. The defined tests can be executed within the *verification and validation* (V&V) activities to discover and repair defects; however, it is not always possible to determine whether the number of defined tests is sufficient and meaningful.

The cost of defect repair is correlated with the development stage in which those defects are detected. The earlier the defect is discovered the lower the cost is. In contrast, defects detected out of phase are increasingly expensive to fix. Moreover, defects detected by customers are the most expensive defects because they may require changing requirements, design, and implementation (O'Regan, 2014).

3.2.5 SQA Vs SQC

Although the terms quality assurance and quality control are often used interchangeably, each one of them comprises different tasks and activities, and it serves different objectives. Software assurance is dedicated to ensuring an acceptable level of quality throughout the whole development process, whereas quality control tends to discover defects and check the conformance of practically completed products. Besides, SQA is prevention-oriented, where SQC is detection-oriented (table 3.2).

The breadth of SQA and SQC is quite different. SQA has a larger scope because a) it tends to be more organizational, b) it encompasses a variety of infrastructures and more additional activities, and c) it focuses on all the development efforts that may occur throughout the software process. On the other hand, SQC includes only the activities that focus on controlling the quality of a single product.

Additionally, SQA is performed during most phases of the software lifecycle starting with the requirements definition, passing by design and coding, and finishing with the product release. However, SQC activities are limited to evaluation and testing phases.

	SQA	**SQC**
Purpose	Preventing causes of errors and flaws by setting up and improving the appropriateness of the software process.	Identifying defects and checking the satisfaction of functional and non-functional requirements.
Focus	Process	Product

Essence	Prevention-oriented	Detection-oriented
When	During the whole process	Before software release

Table 3.2. A comparison between software quality assurance and software quality control

3.3 Error, fault (defect), failure

Software systems are used daily in our life, such as work, home, shopping, etc. However, these systems may not work as expected and sometimes may go to fail. For instance, a system is unable to deliver results at the expected time, or it may suffer from security and reliability issues. The root cause of software failure is errors, which are human mistakes in nature. Subsequently, errors may cause defects, which may cause failures (figure 3.1).

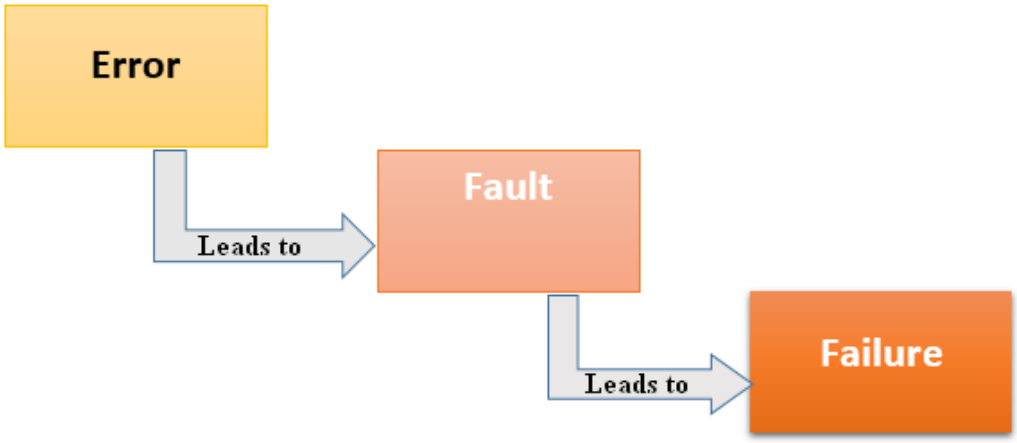

Figure 3.1. The causality relationship between error, fault, and failure

3.3.1 Errors

Errors are the root cause of software failure and poor quality. An error can be a procedure error, a code error, a documentation error, and so forth. The cause of all these errors is mainly human. Therefore, any member of the team represents a potential source of errors. A developer can make logical and calculation mistakes, an architect can make design errors, a business analyst may misunderstand or misinterpret a client requirement, and even a customer can commit errors by providing insufficient or unclear requirements definition. In this section, we will outline some types of errors that may occur at any stage of the development process and cause defects and failures.

- Faulty requirements definition: this type of error is usually committed by the client who may define requirements erroneously, provide incomplete definition, or include unnecessary requirements.
- Defective client-developer (architect, or analyst) communication: this type of error is occurred due to misunderstanding the client's requirements in the early stages of the software lifecycle, or misunderstanding and mishandling the changes of these requirements presented by the client orally or written in forms.

- Deviation from software requirements: developers may deviate from the documented requirements deliberately or unintentionally due to many causes, such as time and budget pressure, or using pre-existed software modules without sufficient analysis and adaptation to new requirements.
- Design flaws: design defects may engender the production of poor software quality. This type of error is caused often by system architects and engineers. Typical errors may include the non-conformance of the designed architecture to the requirements and specified architecture, erroneous task sequencing, erroneous boundaries and conditions definition, and so forth.
- Code errors: this type of errors committed by programmers may include several errors, such as errors in data selection, or errors in manipulating data structures and development tools.
- Non-compliance with the designed architecture: this type of error is due to misunderstanding the design documentation or communication failure between programmers and architects. The non-conformance of the implemented architecture with the designed one is called architecture erosion (chapter 1).
- Shortcomings of the testing process: leaving a greater part of errors undetected or uncorrected is due to the failure of the testing process. This type of error made by testers includes incomplete test plans, documenting and reporting errors failure, correcting faults failure due to the misinterpretation of the causes of the faults, etc.
- Documentation errors: documentation errors affect negatively the development and maintenance processes. This type of error may include design, coding, and requirements errors, which generate in their turns additional errors in further stages of development and maintenance and thus troubling the involved teams.

3.3.2 Software defects

A software fault or defect is the discrepancy between the expected and the yielded results. A defect can cause incorrect functioning of a software system under specific conditions or applications, and rarely the software in general (Galin, 2004). More specifically, a defect (caused by software errors) represents the inability of a software product to comply with the specified requirements, which prevents the software from acting as desired and expected. Generally, defects are discovered by testers and reported to developers and architects to be rectified. It should be noted that not all software errors become defects, for instance, erroneous code may not cause defective software functionalities. In this context, defects can be categorized into a set of severity levels as follows.

a. Critical. A critical defect is a catastrophic fault that affects directly the core functionalities of a software product, and therefore, needs to be treated and removed immediately. This type of defect may lead to collapsing the entire system or the failure of its essential features and functionalities.

b. Major. Major defects do not cause the complete failure of a system; however, they are responsible for affecting the main function of a software product.

c. Minor. Minor defects have less impact on the system and their damage is somewhat slight. The consequences of these defects can be noticed during the execution of the software; however, they do not prevent users from executing the desired task.

d. Trivial. These defects have no impact on the software product and the execution of its functionalities. For instance, grammatical or spelling errors.

The classification of these defects into these four categories is very important in the context of this dissertation. We defined these classes because the proposed evaluation framework MS-QuAAF relies heavily on the severity level of defects to calculate the fulfillment of the specified non-functional requirements and the defectiveness of the relevant software architecture using dedicated metrics. The quantification of the severity of defects by the framework allows enhancing the estimation accuracy of the rate of quality attributes satisfaction, software architecture erosion and deviation, and architecture defectiveness (see chapter 5, 6).

Additionally, defects can be classified from another point of view according to the priority of their treatment due to business and technical factors. As a result, defects can be of high priority and necessitate to be repaired as soon as possible, of medium priority and require to be treated in the next release, or of low priority and their treatment can be performed voluntarily with other defects.

3.3.3 Software failure

A software failure is the result of a software defect that makes the system behave incorrectly and fail to perform as expected in the real environment. However, not all defects conduct to failure that disrupts the use of the system (figure 3.2). A defect becomes a failure only when is activated. In other words, the defect is activated when the user applies a specific faulty application. In certain cases, the defect is never activated and stays inactive in the code. This is possible since a defective system application is never executed by the user or the combination of actions that activate the defect never occurs (Galin, 2004).

To prevent failure, the assessment framework MS-QuAAF focuses on defects that occur in two main stages of the development process, which are design and implementation. It assesses architecture after the accomplishment of these two stages and then generates the defectiveness and conformance reports. The goal is to promoting the quality of software architecture from the beginning of its lifecycle, decreasing design and implementation defects, and thus minimizing the chances of software failure.

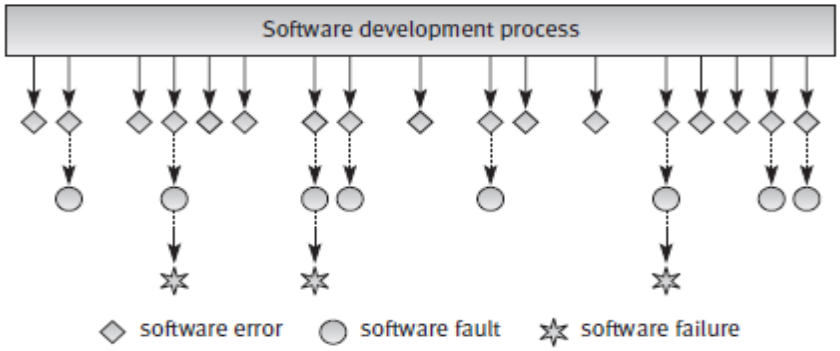

Figure 3.2. Software errors, defect, and failure

3.4 Quality models

The proposition of software quality models is backdated to the 1970s, such as the McCall's and the Boehm's models. The main objective of such models is to provide standardized tools to measure software quality. Each model consists of a set of quality characteristics (quality attributes or factors). Therefore, general and specific software products can be assessed according to these characteristics in order to enhance and achieve the desired quality level. However, choosing which quality model to follow can be a challenging task (Al-Qutaish, 2010). In this section, we will present five of the most well-known software quality models and the comparison between them.

3.4.1 The McCall's quality model

Known as the General Electrics Model, the McCall's quality model is one of the most known quality models in the literature (McCall et al., 1977). It was developed specifically for the U.S. Air Force Electronic Systems Division (ESD) and the Rome Air Development Center (RADC) to provide standards and technical guidance to software acquisition managers. It attempts to bridge the gap between the users' views and the developers' priorities through a set of quality factors and their quality criteria. More specifically, the model consists mainly of a) quality factors, b) each factor is developed hierarchically to a set of quality criteria, and c) for each criterion, a set of programming language independent metrics are defined.

Quality factors are user-related characteristics that contribute actively to the quality of a software system. Therefore, they represent the external attributes of software that reflect the users' view. On the other hand, quality criteria are software-related attributes by which factors are judged and defined, which implies they reflect the internal view (the developer view) of a software system.

The McCall's model consists of exactly *eleven* factors categorized into three major activities as follows (figure 3.3).

- Product revision: this activity encompasses three quality factors, which are maintainability, flexibility, and testability.

- Product operation: this activity incorporates five quality factors, which are correctness, reliability, efficiency, integrity, and usability.
- Product transition: This activity includes three quality factors, which are portability, reusability, and interoperability.

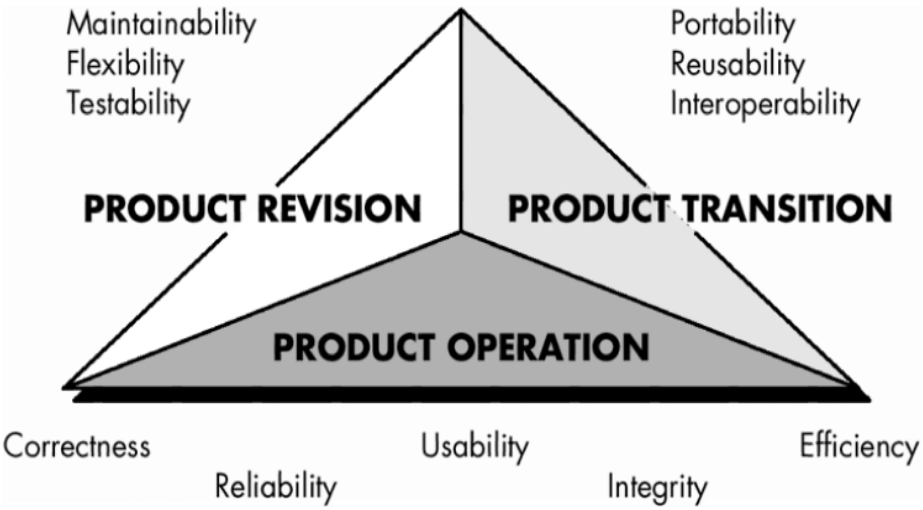

Figure 3.3. The McCall's quality triangle

Each factor consists of at least *two* quality criteria (figure 3.4), and thus the total number of those criteria is *twenty-three*. It should be noted that some criteria are common to more than one factor. For instance, the modularity criterion is common to maintainability, testability, reusability, and interoperability factors. This implies that the McCall's model defines high-level relationships and user-oriented interactions between some of its proposed factors. Furthermore, each criterion is assessed by a set of measurement metrics, in this context, these metrics are computed by answering *yes* and *no* questions. For example, if these questions are answered equally, this means that the quality criterion is achieved by 50% (Al-Qutaish, 2010).

In summary, the McCall's model attempted to provide a complete software quality picture by providing eleven quality factors, twenty-three quality criteria, and measurement metrics for each criterion to cover internal and external attributes of target software systems.

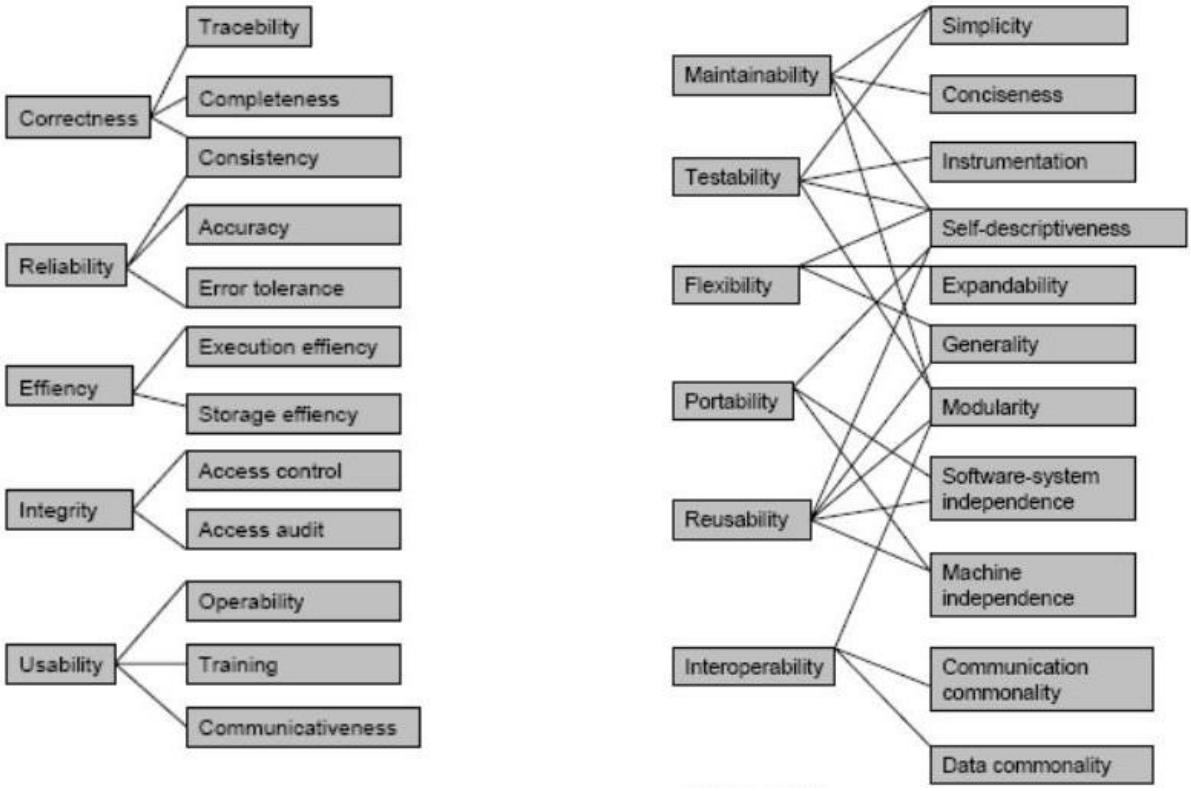

Figure 3.4. The McCall's hierarchical quality model

3.4.2 The Boehm's quality model

In 1978, B.W. Boehm introduced his quality model to assess quantitatively and automatically software quality. Similar to the McCall's model, the Boehm's model defines hierarchically a set of quality characteristics and metrics that contribute to the overall software quality (B. W. Boehm et al., 1978; Barry W. Boehm et al., 1976). However, these characteristics are grouped into three levels as follows (figure 3.5).

- High-level characteristics: at this high level, there are three primary characteristics of the system stated as follows.
  - As-is utility: the extent to which we can use the software product as-is.
  - Maintainability: the effort required to understand, modify, and retest a software product.
  - Portability: the effort required to change the system to fit with the new environment.
- Intermediate-level: this mid-level encompasses seven quality characteristics that altogether denote the quality expected from a software system.
  - Portability (the portability characteristic is common between two levels).
  - Reliability: extents to which the software can perform its intended functions.
  - Efficiency: the total of resources required to execute the intended functions.
  - Usability: the effort required to learn, understand and operate the software.
  - Testability: the effort required to check that the system performs its intended functions.

- Understandability: the degree of clarity of the system's structures and behavior.
- Modifiability: the effort required to modify the system during maintenance.

- Low-level: this level contains primitive characteristics that provide a foundation for defining metrics. Each mid-level characteristic is divided into two or more primitive characteristics. For instance, the modifiability characteristic is divided into *structuredness* and *augmentability*. Each primitive characteristic in this model is measured by at least one metric. Boehm defined a metric as "measure of extent or degree to which a product possesses and exhibits a certain (quality) characteristic" (B. W. Boehm et al., 1978).

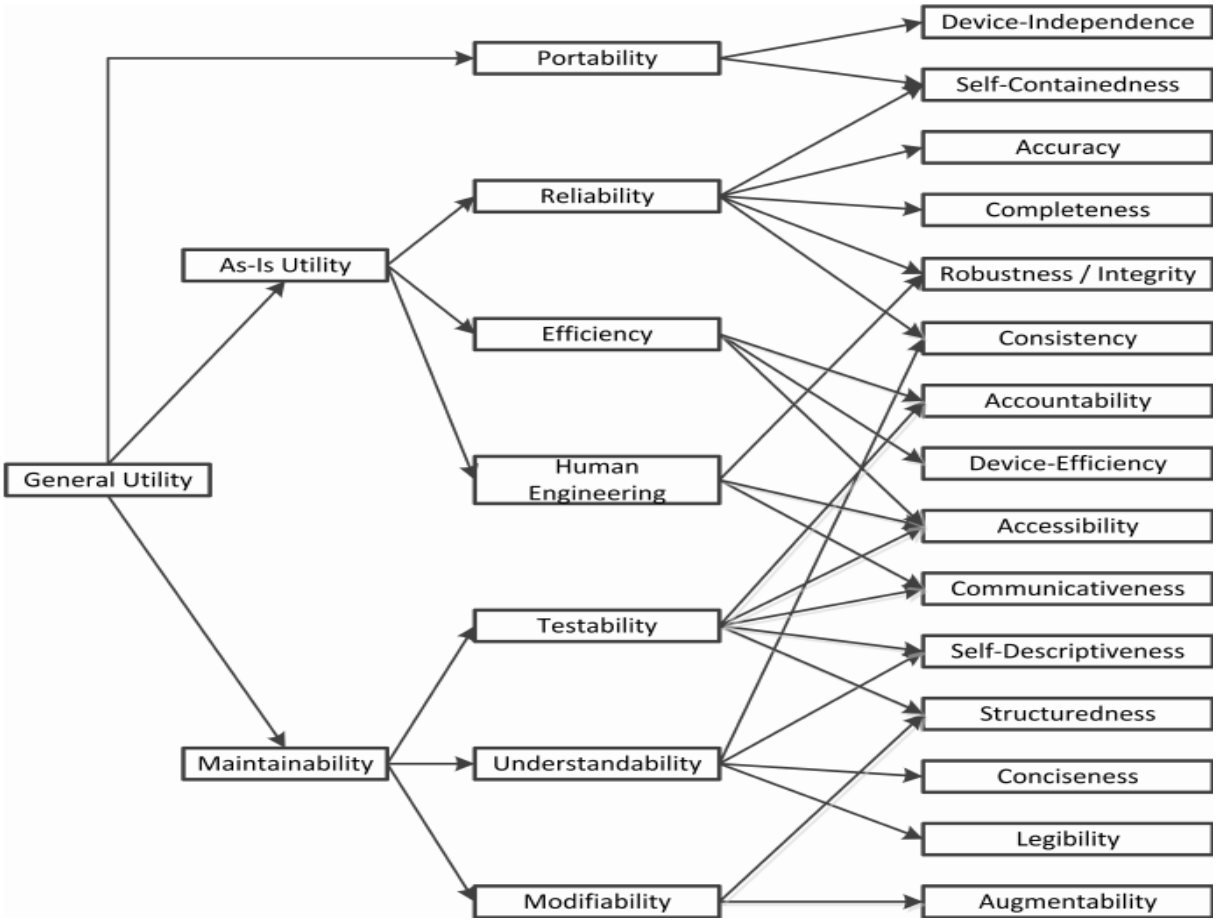

Figure 3.5. The Boehm's quality model

3.4.3 The Dromey's quality model

Dromey proposed an evaluation framework that allows constructing software quality models in a structured and systematic way by proceeding from the tangible to the intangible (Dromey, 1995). He defined a generic model for building quality into the software product. This model consists of three main entities: components (variables, functions, modules, requirements, etc.), quality-carrying properties (tangible properties) for components, and high-level quality attributes. The framework evaluates the quality of software components through the measurement of their quality-carrying properties. The latter are divided into four basic

categories: correctness properties, structural properties, modularity properties, and descriptive properties (documentation). Each category of quality-carrying properties is related directly to high-level quality attributes that characterize the quality of a software system (figure 3.6). These attributes are taken from the International Standard ISO-9126 (replaced by ISO/IEC 25010:2011 (*ISO 25010*, n.d.)). However, Dromey added to this list, the *reusability* attribute.

The Dromey's model is a product-based model that distinguishes that quality evaluation differs from one product to another. It provides a process that permits building quality-carrying properties into these products. In other words, the model establishes the link between tangible characteristics and less tangible quality attributes. Furthermore, the model can assist in searching, detecting, and classifying quality defects caused by the violated properties. This information allows building a comprehensive set of defects for any language environment (Dromey, 1995).

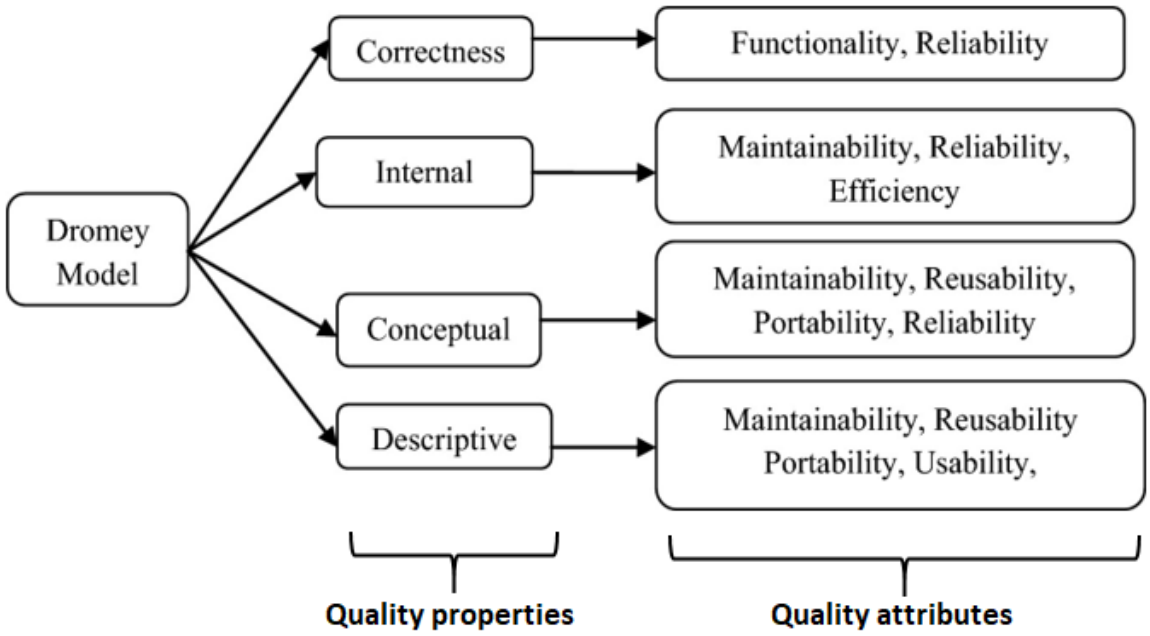

Figure 3.6. The Dromey's quality model

3.4.4 The FURPS quality model

The FURPS model was developed originally by Robert Grady at the Hewlett-Packard (Grady, 1992). This model stands for five quality characteristics divided into functional (**F**) and non-functional requirements (**URPS**) as follows (figure 3.7).

- **F**unctionality: this attribute is evaluated by assessing feature sets, software capabilities, and overall system security.
- **U**sability: this attribute includes human factors, user interface consistency, aesthetics, training materials, and documentation.

- **R**eliability: this attribute can be evaluated by measuring severity and frequency of failure, predictability recoverability, MTTF (mean time to failure), etc.
- **P**erformance: this attribute can be evaluated by measuring response times, resource consumption, recovery times, efficiency, etc.
- **S**upportability: this attribute may include adaptability, testability, installability, maintainability, serviceability, configurability, compatibility, and extensibility.

This model was later extended to FURPS+ by IBM Rational Software (Jacobson et al., 1999; Kruchten, 2000). The "+" indicates more requirements and standards, such as design constraints, interface requirements, implementation requirements, and physical requirements.

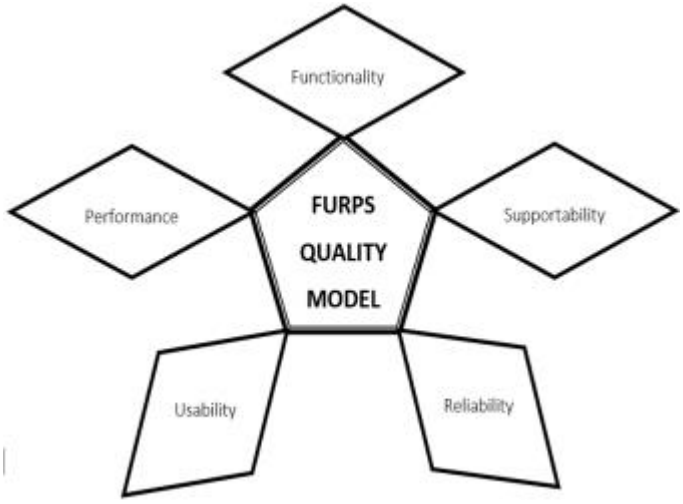

Figure 3.7. The FURPS's quality model

3.4.5 The ISO/IEC 25010:2011 Quality Models

The product quality model of the Systems and software Quality Requirements and Evaluation (SQuaRE) was developed specifically to identify quality attributes that can be used to establish stakeholders' requirements, their satisfaction criteria, and their corresponding measures (*ISO 25010*, n.d.). It was derived from the ISO/IEC 9126:1991, which was developed to support these needs. The latter replaced by the ISO/IEC 9126:2001, which defines six quality attributes and *twenty-seven* sub-attributes for internal and external qualities (Al-Qutaish, 2010). Furthermore, this standard defined a quality in use model, which includes four quality characteristics. However, the ISO/IEC 9126:2001 was also withdrawn and revised by the ISO/IEC 25010:2011 (figure 3.9). The latter integrates the same quality characteristics with some adjustments:

- Adding Security as a characteristic instead of a sub-characteristic of functionality.
- Adding compatibility as a characteristic.
- Removing the Compliance sub-characteristics.
- Adding new sub-characteristics, such as availability, modularity, and reusability.
- Giving more accurate names to characteristics and sub-characteristics.

- Adding Context Coverage as a new characteristic to quality in use.

This standard defines two models relevant to both computer systems and software products:

a) A quality in use model. This model incorporates *five* quality characteristics and *nine* sub-characteristics that concern the impact of interaction when using software products and computer systems in a particular context of use. The ISO/IEC defined the quality in use as "the degree to which a product or system can be used by specific users to meet their needs to achieve specific goals with effectiveness, efficiency, freedom from risk and satisfaction in specific contexts of use" (*ISO 25010*, n.d.). Accordingly, the following quality characteristics are defined (figure 3.8):
   - Effectiveness.
   - Efficiency.
   - Satisfaction: this attribute encompasses Usefulness, Trust, Pleasure, and Comfort.
   - Freedom from risk: this attribute includes Economic risk mitigation, Health and safety risk mitigation, and Environmental risk mitigation.
   - Context coverage: this attribute incorporates Context Completeness and Flexibility.

b) A product quality model. This model incorporates *eight* quality characteristics and *thirty-one* sub-characteristics related to static and dynamic properties of software products and computer systems (figure 3.9):
   - Functional Suitability: it includes Functional completeness, Functional correctness, and Functional Appropriateness.
   - Performance Efficiency: it includes Time Behavior, Resource Utilization, and Capacity.
   - Compatibility: it encompasses Co-existence and Interoperability.
   - Usability: it encompasses Appropriateness Recognizability, Learnability, Operability, User Error Protection, User Interface Aesthetics, and Accessibility.
   - Reliability: it incorporates Maturity, Availability, Fault Tolerance, and Recoverability.
   - Security: it includes Confidentiality, Integrity, Non-Repudiation, Authenticity, and Accountability.
   - Maintainability: it comprises Modularity, Reusability, Analyzability, Modifiability, and Testability.
   - Portability: it comprises Adaptability, Installability, and Replaceability.

According to this standard, these models are developed for supporting specification, development, requirements, maintenance, and quality assurance and control of software products and computer systems. It should be noted that these quality models have excluded functional properties; however, they included the Functional Suitability characteristic (*ISO 25010*, n.d.).

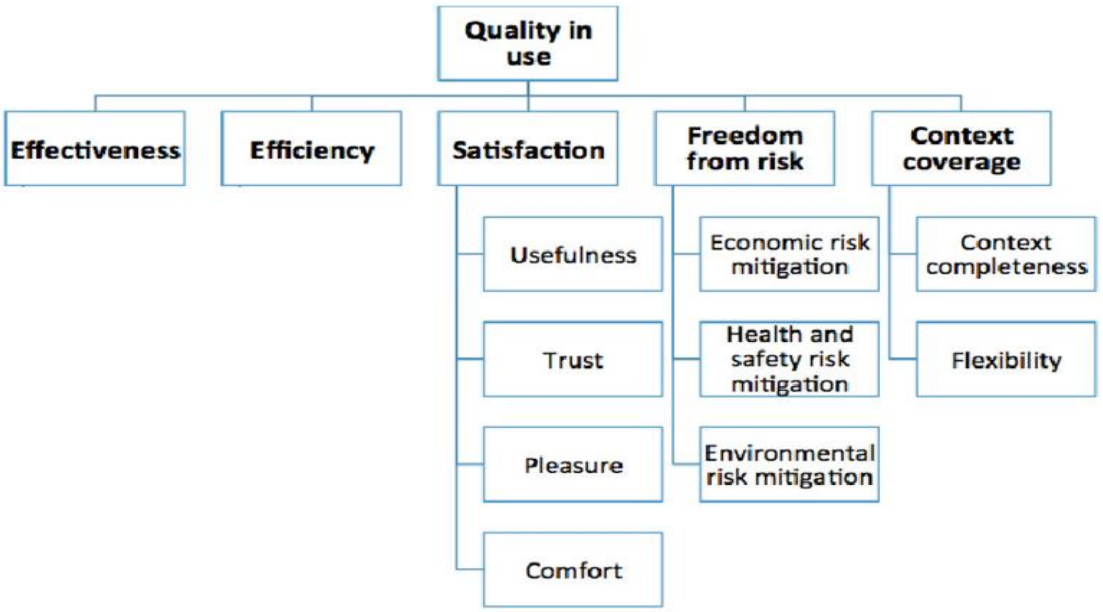

Figure 3.8. A quality in use model (ISO/IEC 25010:2011)

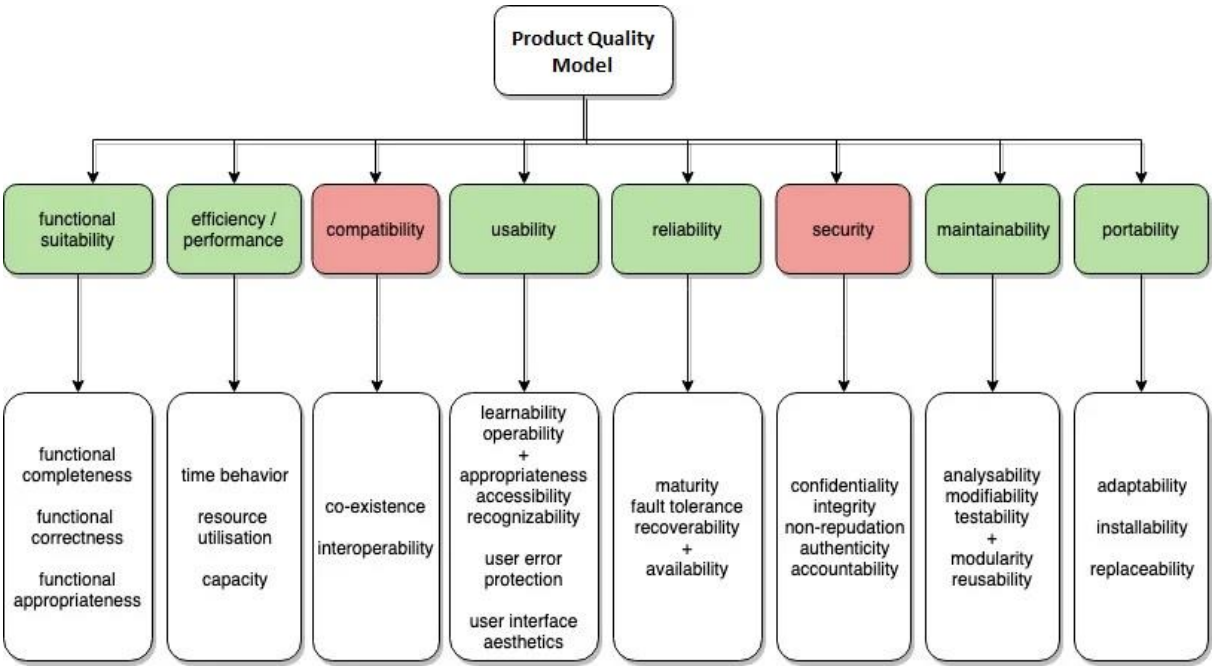

Figure 3.9. A product quality model (ISO/IEC 25010:2011)

3.4.6 Analysis and discussion

By analyzing and comparing the quality factors (attributes, characteristics) of the above models, we found that only one quality characteristic is common to all these models, which is the Reliability attribute (Table 3.3). The same result is found by Al-Qutaish (Al-Qutaish, 2010) in his comparative study; however, he utilized the ISO 9126 instead of ISO 25010. This result

demonstrates that Reliability is the most important quality characteristic in software engineering according to these models.

Quality characteristics	McCall's Model	Boehm's Model	Dromey's Model	FURPS Model	ISO/IEC 25010:2011
Correctness	X				
Maintainability	X		X		X
Testability	X	X			
Reliability	X	X	X	X	X
Efficiency	X	X	X		X
Usability	X		X	X	X
Integrity	X				
Portability	X	X	X		X
Interoperability	X				
Reusability	X		X		
Flexibility	X				
As-is Utility		X			
Human Engineering		X			
Understandability		X			
Modifiability		X			
Functionality			X	X	
Performance				X	X
Supportability				X	
Functional Suitability					X
Compatibility					X
Security					X

Table 3.3. Comparison between the presented quality models

Furthermore, Table 3.3 illustrates that three quality characteristics are included in four quality models, which are Efficiency, Usability, and Portability, and one characteristic is included in three models, which is Maintainability. These four attributes are also considered of great importance in assessing software systems. On the other hand, some attributes are defined exclusively by one quality model, such as Supportability and Understandability. However, this does not diminish their role in quality evaluation. Actually, these attributes may be considered important in specific contexts by specific stakeholders. Besides, we found in this comparison that some characteristics in some models are defined as sub-characteristics in other models, for instance, the Integrity and Interoperability attributes in the McCall's model are defined as sub-attributes of the Security and Compatibility attributes in the ISO/IEC 25010:2011, respectively.

To this end, we found that the McCall's, the Dromey's, and the ISO/IEC 25010:2011 models are very close to each other in terms of the defined quality characteristics. More specifically, they have five quality attributes in common, which are Maintainability, Reliability, Efficiency, Usability, and Portability. The Boehm model can also be considered close to these models but to a lesser extent. It should be noted that all the attributes in common belong to the McCall's

model, which proves that the latter is the foundation on which most of the successor models are built, even the recent ones, such as the ISO/IEC 25010:2011.

3.4.7 MS-QuAAF and Quality models

MS-QuAAF is a quantitative evaluation framework that assesses software architecture with respect to stakeholders' NFR goals (chapters 5, 6). These goals that represent quality characteristics may belong potentially to a specific quality model. Therefore, among the questions that may arise when using this framework are the following.

- What is the relationship between the framework and quality models?
- Do quality models have the same impact on the evaluation effort in terms of time and money?

One of the major characteristics of MS-QuAAF is its independency of quality models. The framework has no knowledge about the quality model of the quality attributes under assessment. The evaluation consists of accepting any quality attribute as input, calling the evaluation services to perform the assigned assessment tasks, and then producing the evaluation report as output. Therefore, the evaluation effort of target architectures is always performed independently of their quality models, which means that the framework supports all models.

For the second question, if we suppose that many quality models are fully adopted by evaluators in the assessment process of the same architecture, then the evaluation effort in terms of time and money is obviously not the same. This is due to the complexity of quality models and the number of their quality characteristics and sub-characteristics. For instance, the evaluation efforts of the same attribute A_1 that belongs to the quality models M_1 and M_2 may not be equal because A_1 of M_1 has sub-attributes more than $A_1$of M_2.

3.5 Conclusion

This chapter tackled the key topic of this dissertation, which is software quality. First, we have started by depicting the concept of quality in general according to the point of view of the most known figures of the quality movement, such as Deming, Juran, and Shewhart. Their definitions of quality are divided into two major philosophies. The former defines quality as conformance to requirements, in which a product is judged of high quality if it meets the specified requirements. The latter defines quality as customer satisfaction, in which a product is judged of high quality if it fulfills customer's needs and expectations. Furthermore, we have presented our perspective on software quality, which is the degree of achievement of the desired quality attributes of all stakeholders and not just customers, taking into account the tradeoffs among these attributes. This definition is the basis on which the framework MS-QuAAF was built.

Second, we have outlined three major quality management approaches developed to satisfying customer's needs, as well as the organization's objectives, which are Total Quality Management, Software Quality Assurance, and Software Quality Control. These approaches

represent quality management philosophies and activities employed to manage quality within an organization.

Third, we have depicted the problem of software errors and defects that usually cause software quality degradation. In this context, defects are mainly caused by human errors, such as faulty requirements definition or code errors. Defects can be critical, which may conduct to software failure and make the system fail to perform as expected in the real environment.

Fourth, we have presented some of the most well-known quality models, such as the McCall's model and the ISO/IEC 25010:2011 model. These models provide standardized measures to assess software quality by defining a set of quality characteristics and sub-characteristics. Furthermore, we have explained the relationship between quality models and the framework MS-QuAAF, and the impact of these models on the assessment process.

Chapter 4

Software architecture analysis and evaluation methods

In this chapter, we will depict the main evaluation methods proposed to assess software architecture. We classified these methods according to a set of criteria, such as measurability and evaluation techniques. The classification criteria have facilitated categorizing these methods, comparing their evaluation techniques, and emphasizing their advantages and drawbacks. Performing this literature study and extracting the main shortcoming of evaluation methods allowed us at the first time to construct the skeleton of our evaluation framework MS-QuAAF and define its main goals. Subsequently, we have developed and improved the framework gradually until its final version, which will be presented in detail in the next chapters.

Content

4.1 Motivation for architecture evaluation 56
4.2 Classification criteria for evaluation methods 57
4.3 Evaluation methods 59
4.4 A concluding discussion about the presented evaluation methods 89
4.5 Conclusion 91

4.1 Motivation for architecture evaluation

Since it has emerged in the last decade of the 20th century as a sub-discipline of software engineering, software architecture (SA) has been considered as the appropriate level to deal with quality attributes (Dobrica & Niemela, 2002). Hence, it has attracted the attention of researchers, as well as practitioners since the size and complexity of software systems have been augmented tremendously along with the increased demand of stakeholders' to satisfying their quality requirements (figure 4.1). They have realized (researcher and practitioners) that high-level design description is the key to understanding and managing complex software systems. Moreover, they recognized that the achievement of quality attributes is constrained by the SA of these systems and the architectural decisions made at this level (Babar et al., 2004).

Undoubtedly, SA plays a major role in the achievement of the desired quality attributes; therefore, analyzing and assessing SA against those attributes of concern is very important as early as possible. In this context, the main goal of SA evaluation is to detect architectural defects

and liabilities that may degrade the quality of architecture, and thus determining the extent to which architectural decisions meet the quality requirements. More specifically, the early evaluation permits:

- Identifying and mitigating design risks.
- Predicting the quality of the system before it has been built.
- Verifying that quality requirements are addressed in the design.
- Assessing the potential of the architecture to deliver a system that is capable of fulfilling those requirements.
- Reducing the development effort, and saving the integration, testing, and maintenance costs by fixing defects at the initial stages of the software development.

Although most of the assessment methods focus on the early evaluation, other methods gave attention to evaluating architecture at late stages of development, generally after the implementation stage. The major aim of the late evaluation is to verify the conformance between the intended architecture and the implemented one. In other words, verifying that architectural decisions made at design time, such as tactics and architectural styles are not violated at the implementation stage. The infringement of these decisions causes the phenomenon of architecture erosion, which can be defined as the mismatch between the descriptive architecture (the realized one) and the prescriptive architecture (the intended one) (de Silva & Balasubramaniam, 2012). However, most of the well-known evaluation methods and frameworks in the literature focus on the evaluation at the early stages, which will be addressed extensively in the current chapter.

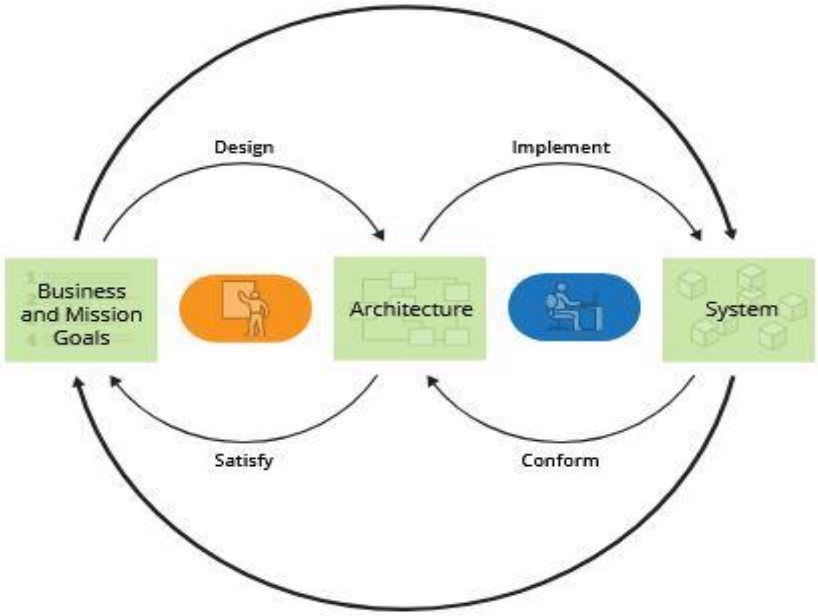

Figure 4.1. The relationship between software architecture and business goals

4.2 Classification criteria for evaluation methods

Different communities with different points of view have developed a considerable number of techniques to evaluate SA with respect to the targeted quality attributes. The most well-known

evaluation methods are scenario-based, software metrics, attribute model-based, and machine learning-based. The categorization of these methods can be performed according to many criteria. Dobrica (Dobrica & Niemela, 2002) has defined a framework for classifying scenario-based evaluation techniques according to a set of elements: specific method's goal, evaluation techniques, quality attributes, SA description, Stakeholders' involvement, method's activities, reusability of an existent knowledge base, and method validation. Additionally to these elements, Babar et al. (Babar et al., 2004) have added other categorization elements for his classification framework, such as maturity stage, process support, and tool support. Sobhi et al. (Sobhy et al., 2021) have defined four criteria for classifying methods that evaluate SA under uncertainty, which are quality evaluation, quality attributes consideration, level of autonomy, and uncertainty management.

In this chapter, we will categorize evaluation methods according to six classification criteria:

a. **Measurability (C1):** Measurability allows dividing evaluation methods into two basic classes, which are qualitative and quantitative evaluation. C1 permits distinguishing methods that use qualitative techniques to assess SA from methods that use measuring (quantitative) techniques for the same purpose. However, some methods are called hybrid because they combine both techniques.
b. **Evaluation technique (C2):** This criterion permits determining the evaluation techniques within each main class (Qualitative or quantitative). Qualitative methods consist mainly of questioning techniques to generate qualitative questions to be asked of an architecture. These techniques include questionnaires, checklists, and scenarios. On the other hand, quantitative techniques suggest quantitative measurement to evaluate an architecture. using, for example, metrics, simulations, and experimentations.
c. **Evaluation stage (C3):** This criterion permits to classify evaluation techniques according to the development stage in which architecture evaluation is performed. The latter concerns principally the designed architecture (design stage), the implemented (realized) architecture (implementation stage), or both of them (continuous evaluation).
d. **The number of quality attributes (C4):** This criterion permits to determine how many (single or multi-attributes) and which quality attributes are covered by each technique.
e. **Tradeoffs (C5):** this criterion determines whether an evaluation technique supports tradeoffs among the competing quality attributes.
f. **Tool support (C6):** This criterion allows determining whether an evaluation method provides tool support.

According to these criteria (figure 4.2), we divided the evaluation methods into three main categories:

- Scenario-based methods: Most of these methods are qualitative, whereas the rest of them are hybrid.

- Goal decomposition analysis methods: The proposed methods are qualitative, quantitative, or hybrid.
- Metric-based methods: These methods are based on metrics, which means they are purely quantitative.

In this chapter, we have chosen to present these categories because a) they have been addressed excessively in the literature, especially scenario-based and metric-based, b) they address three aspects related directly to our assessment framework, which are architecture analysis, architecture evaluation, quality attributes, and metrics. Methods that use techniques, such as machine learning and statistical algorithms will not be presented in this dissertation and they are considered out of scope

4.3 Evaluation methods

There are numerous evaluation methods in the literature proposed to evaluate software architecture, each of which has its specificity, advantages, and drawbacks. In this chapter, we have chosen to portray a) the most addressed methods in the literature, and b) methods that are somehow related to our evaluation method. More specifically, three categories of evaluation methods will be presented in this section, which are scenario-based methods, goal decomposition methods, and metric-based methods. Subsequently, these methods will be compared and classified according to the above classification criteria.

4.3.1 Scenario-based methods

At the beginning of the 1990s, scenario-based frameworks appeared as the descriptive means for specifying and assessing quality attributes within a context. The latter means that quality attributes cannot exist in isolation; therefore, they must be evaluated with respect to a specific context. For example, evaluating the security attribute with respect to some types of threats, or performance with respect to specific resource utilization, and so forth. A scenario can be defined as a specified sequence of steps that may involve the use or modification of the system (Kazman et al., 1996). The main goal of this type of evaluation is to verify the reaction of architecture against the specified scenarios to decide its wellness within specific contexts.

There is a myriad of scenario-based methods proposed for architecture analysis and evaluation. Therefore, we will attempt to present the most known scenario-based methods within the following non-exhaustive study, in which diverse evaluation techniques will be depicted. The reason behind presenting this category of evaluation methods is that they focalize mainly on architecture analysis at the design stage, which allows discovering defects as early as possible. This inspired us to define one of the primary services of MS-QuAAF that allows analyzing architecture against the documentation specified at early stages. However, MS-QuAAF's service evaluates architecture quantitatively through a set of dedicated metrics instead of using scenarios within qualitative techniques.

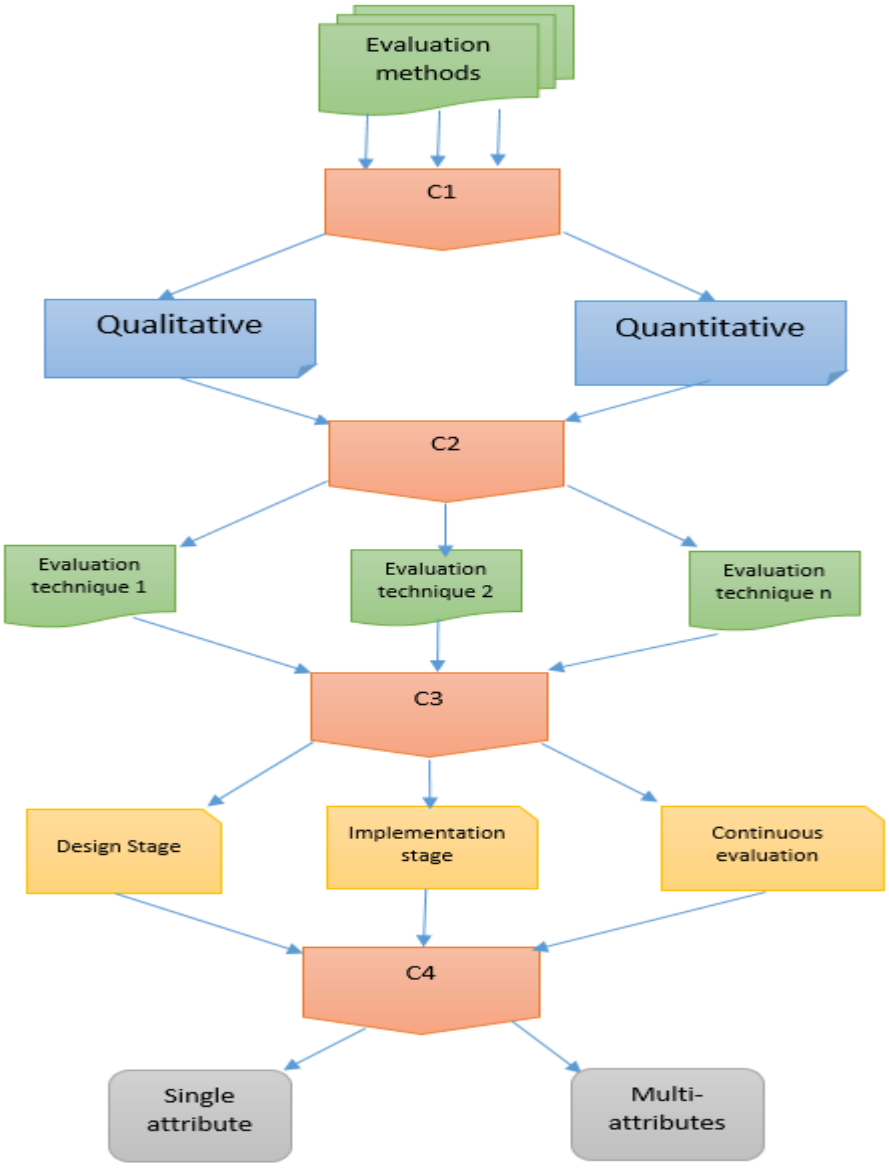

Figure 4.2. Classification criteria for evaluation methods

4.3.1.1 Scenario-based Architecture Analysis Method (SAAM)

SAAM is the first well-known scenario-based architecture analysis method proposed originally to allow comparison between the competing architecture solutions (Kazman et al., 1996). Over the years, the method's steps have evolved as a result of the increased author's experience with architecture analysis. SAAM is interested in evaluating the architecture to determine its fitness with respect to certain system properties (qualities). More specifically, its main goal is to evaluate the architecture design decisions by checking architectural principles and assumptions against the documentation that describes the desired properties using a set of scenarios. Additionally, it contributes to identifying risks in software architectures.

Scenarios are brief descriptions of anticipated or expected use of a system from the stakeholders' viewpoints (developers, end-users, etc.).They have been widely used as a technique during a) requirements elicitation, b) design stage to compare design alternatives or

c) development stage to verify the responses of an already-built system to specific operational situations. However, SAAM is considered as the first method that uses brief scenarios (one sentence long) for architecture quality analysis. Scenarios are used to express particular instances of each stakeholders' quality attributes, and then target architectures are analyzed against the constraints specified by each defined scenario to assess the extent to which the architecture satisfies those attributes. It should be noted that this method is destined originally to assess modifiability; however, other attributes, such as portability and extensibility may also be assessed (Sobhy et al., 2021).

The architecture analysis carried out by SAAM consists of the following steps:

- **Describing candidate architecture:** architectures should be described with a clear syntactic architectural notation that all stakeholders involved in the analysis process can understand. An architecture is described in terms of functional partitioning (a collection of the system's functions), structures (components and connectors), and the allocation from functionalities to those structures.
- **Developing scenarios:** scenarios are developed to illustrating the activities that the system must accomplish or anticipating the changes that may be made to the system over time. These scenarios must cover all major and important uses of the target system. SAAM requires the presence of all stakeholders, such as developers, maintainers, and customers to identify the possible scenarios, which represent tasks related to each role of these stakeholders. The scenarios development and architecture description processes are related and iterative (figure 4.3). The final version of the architecture description and scenarios are considered as input to the subsequent analysis activities.
- **Performing scenarios evaluation:** for each scenario, changes to architecture are listed and the cost for performing these changes is estimated. A modification means the introduction of a new component or connector or changing the specification of an existing component or connector. At the end of this step, a summary table that lists all scenarios is created. The description of the impact or the changes that will be made by each scenario is also included in this table. The description illustrates the components and connectors that must be changed and the new components and connectors that must be introduced. This table is also useful for comparing candidate architectures to select which architecture can support better the list of scenarios.
- **Revealing scenarios interaction:** this process consists of determining scenarios that affect common sets of components. This case is defined by SAAM as scenario interaction. Measuring interaction allows determining the extent to which separation of concerns is supported by the target architecture. Architectures with fewer scenario conflicts are favored par SAAM.

- **Overall evaluation:** Scenarios and their interactions are weighted according to their relative importance, which allows determining the overall ranking. This process necessitates the involvement of all the system's stakeholders.

SAAM has been validated in various domains and case studies; therefore, it was considered as a mature evaluation method. Among these case studies, we can find a global information system, air traffic control, and a revision control system (WRCS).

4.3.1.2 Extending SAAM by integrating in the domain (ESAAMI)

ESAAMI is a combination of analytical techniques provided by SAAM and reuse-based approaches. Hence, SAAM is integrated and deployed in domain-specific and reuse-based development processes (Molter, 1999). From the author's viewpoint, reducing risk consists of reusing architectures from early successful projects in the same application domain. This can be achieved if the reused architecture and the system under development are adequately similar. Additionally, the reused architecture must provide flexibility to be adapted and fitted with the new system.

This method is similar to SAAM concerning SA description, evaluation techniques, and stakeholders' involvement. However, ESAAMI incorporates already existing knowledge, which makes the development process reuse-aware. Making available this knowledge requires providing reusable assets, such as proto-scenario, proto-evaluation, and architectural hints. These assets constitute analysis templates. A template in a domain-oriented development can be created on an abstraction level defined by the commonalities of the applications in this domain. In this context, proto-scenarios represent a generic description of reuse situations or interactions with the system. They are designated to be used in the stage of scenario elicitation, in which the selected scenarios are refined to fit specific details of the target application. The evaluation of these scenarios is facilitated by early analysis's protocols of different projects as well as proto-evaluation. The latter can describe, for instance, how scenarios can be executed using a set of abstract architectural elements. In this regard, the proto-evaluation can be refined to taking into account the actual architecture elements and their relationships. In addition to proto-evaluation and protocols, hints are associated with scenarios to indicating which architectural elements would make the scenario convenient to handle (Molter, 1999). Similarly, scenarios interaction can also benefit from the early analysis's protocols to characterize, for instance, specific scenarios as typical and necessary.

As a means to compare several architectures regarding their usability in a specific application, scenarios and scenarios interaction can be weighted to depict their impact on this application.

The obtained weights can be reused in different projects in the same domain to make the analysis results comparable.

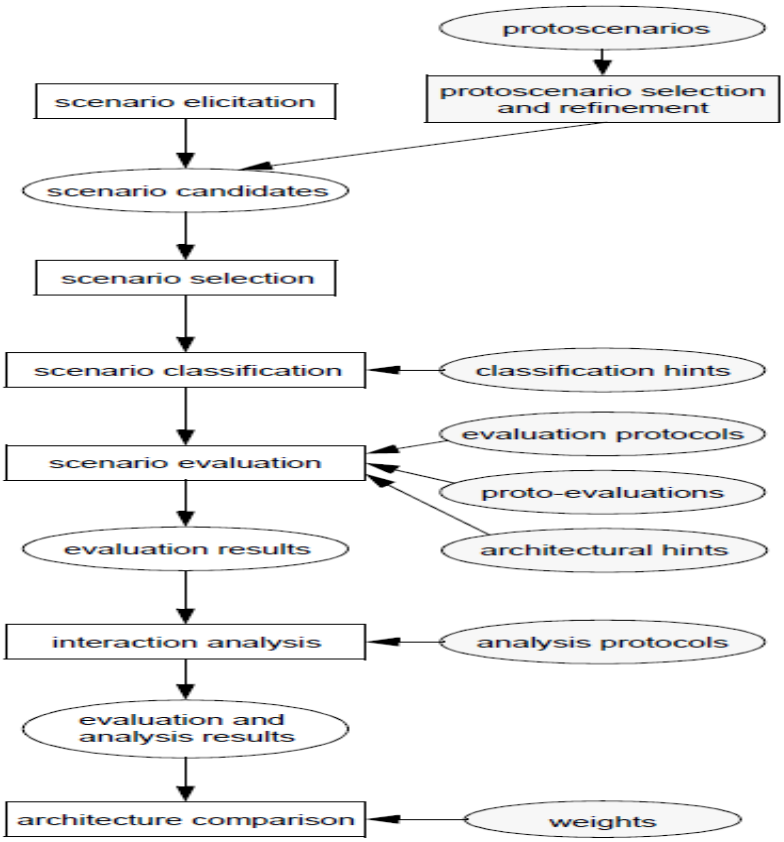

Figure 4.3. ESAAMI approach

ESAAMI allows reusing templates analysis to reduce the cost of analysis and expedite its implementation because of the availability of a suitable foundation for the scenarios elicitation and refinement. In other words, reusing templates in a domain-centric development process allows exploiting experiences and already existed knowledge to boost up the relevance of the analysis results. Besides reusing templates in several projects may conduct to obtain comparable analysis results, which helps in increasing the understanding of the differences between the considered architectures.

The method is not considered validated since the author did not publish records that indicate the application of the method in experimental validation.

4.3.1.3 Software Architecture Analysis Method for Evolution and Reusability (SAAMER)

The authors presented a framework and a set of architectural views to assess the evolution and reuse quality attributes of software architecture by following an analysis approach based on SAAM. To guarantee that the analysis is scientific and well organized, a framework for gathering information and analysis is formulated. The latter has defined four main activities performed iteratively as follows (Lung et al., 1997).

Gathering. In this phase, four categories of information are collected and compiled to perform the analysis, namely stakeholders, architecture, quality, and scenarios information.

Modeling. Modeling is a critical phase because if it is not accomplished correctly, then the whole analysis can be misled. The information collected previously is mapped into usable artifacts. This information is used to capture architecture, driving analysis, and providing feedback in the latter phases. In this phase, the modeling should take into account two aspects, which are the breadth and depth of analysis. The former describes the relationships between scenarios, quality attributes, stakeholders' objectives, and architectural objectives. The latter deals with the abstraction levels at which various categories of information are represented. The depth of analysis impacts its accuracy.

Analyzing. In this phase, SAAM is extended to perform quality attributes analysis, in which scenarios are mapped to SA. The analysis is performed through artifacts, such as domain models (used to compare competing architectures), relevant architectural views, scenarios, and tradeoffs.

Evaluating. In this phase, architecture issues, such as performance bottlenecks are located, and mitigation strategies are enumerated.

In addition to this framework, SAAMER adopts various architectural views that are considered critical for the analysis:

- **Static view:** This view depicts the overall topology, which includes a module diagram, structure diagram, logical diagram, and object diagram.
- **Map view:** This view identifies design violations and the mapping between functions or features and components.
- **Dynamic view:** This view depicts the behavioral aspects of the system. It may include a causal diagram, functional diagram, object interaction diagram, Petri net, etc.
- **Resource view:** This view is interested in the utilization aspect of the system resources.

In this method, scenarios are the main driver for the architecture analysis and the capture of other architectural views. These views allow revealing deeper information, whereas scenarios can describe important functionalities that the system must support where the system needs to be changed. SAAMER extends SAAM by identifying for each scenario, required changes and estimating the necessary effort to make those changes. This information allows determining how the system can support stakeholders' objectives or identifying the risk level or reuse across the application. SAAMER can analyze and expose architecture areas that are tailorable, reusable, and not reusable by using insights views and scenarios rather than developing the architecture from scratch, which may contribute to reducing the time and development effort of the new architecture.

Furthermore, this method answered the question concerning when to stop scenarios generation (unlike SAAM). Two methods are used here. First, the generation is tied to various stakeholders, quality, and architectural objectives. Based on these objectives and domain experts, scenarios are identified and clustered to make sure each objective is well covered. Second, the balance of scenarios regarding objectives is validated using QFD (Quality Function Deployment). In this context, matrices are generated to depict the relational strength from stakeholder and architectural objectives to quality attributes. Subsequently, these attributes are translated into a set of scenarios to reveal the coverage of each quality attribute using an imbalance factor. The latter is calculated by dividing the coverage by quality priority. If the result is less than 1, then more scenarios are needed to cover the concerned attribute.

Concerning the method validation, SAAMER was applied to one system, which is a telecommunication and switching system.

4.3.1.4 The Architecture Trade-off Analysis Method (ATAM)

ATAM is the most popular architecture evaluation method (Sobhy et al., 2021). Its techniques are inspired by three areas: the quality attribute analysis, the notion of architectural styles, and SAAM. ATAM is considered as an evolved version of SAAM, in which the main purpose is not only evaluating the consequences of architectural decisions with respect to quality attributes requirements (Kazman et al., 2000) but to focus on how these attributes interact with each other and tradeoff against each other. In this context, taking decisions to promote one quality attribute may affect negatively other quality attributes. For instance, to promote *availability* and *performance* in a client-server architecture, architects decide to increase the number of servers. However, this decision may affect the *security* attribute by increasing the number of attack points and failures.

The ATAM process consists of nine steps, in which the time required to carry out the analysis depends on the size of the system, the maturity of architecture, and the state of its description.

- Step 1: In this step, the method is presented and described to stakeholders, particularly, architecture team and customer representatives. More specifically, the presentation describes the steps briefly, the procedures that will be for elicitation/ analysis, and the evaluation outputs.
- Step 2: In this step, the project manager describes at a high abstraction level the system to be evaluated to all participants from a business perspective. Principally, the manager describes the most important functional requirements, business goals and motivation, constraints, and architectural drivers (major quality attributes, such as availability and performance).
- Step 3: The proposed architecture will be described with a focus on how it addresses the business drivers. The description should cover technical constraints (hardware, OS,

middleware, etc.), systems with which the system may interact, and architectural approaches that must be followed to meet quality requirements.

- Step 4: This step consists of capturing and identifying architectural (without analyzing) approaches and architectural styles adopted to address the highest priority quality attributes.
- Step 5: In this step, the most important quality goals are identified, prioritized, and refined. The objective of this step is to draw stakeholders' attention to focus on the most critical aspects of architecture by means of building a utility tree. The output of the tree generation is a prioritization of specific attributes realized as scenarios. The obtained prioritized list guides the remainder of the analysis by providing the evaluation team with information, such as where to probe architectural approaches, their risks, tradeoffs, etc. Moreover, this tree allows concretizing the quality attribute requirements and forcing the involved stakeholders to define their quality requirements more precisely.
- Step 6: Based on the high-priority quality goals identified in the previous step, the elicitation and analysis of the architectural approaches that address those goals are performed. The main outputs of this step are the list of architectural approaches, questions associated with them, and responses to these questions. Additionally, architectural risks, sensitivity points, and tradeoffs are generated.
- Step 7: This step is dedicated to brainstorming and prioritizing scenarios. According to the authors, generating scenarios can facilitate discussion and brainstorming when considerable numbers of stakeholders are involved in the ATAM. Based on scenarios generated in the fifth step, a broader set of scenarios is elicited from all stakeholders. Subsequently, the obtained set of scenarios is prioritized via a voting procedure.
- Step 8: This step reiterates step 6; however, only the highly ranked scenarios (from the previous step) mapped to architectural approaches are considered to be test cases for the analysis. This step represents a testing activity that would allow revealing new additional information, such as sensitivity points and tradeoffs. If new information is uncovered, then the utility tree generated previously is failed. Therefore, the evaluation team should return back to step 4, 5, and 6 and work through them until no further new information is revealed.
- Step 9: This is a reporting step, in which the ATAM team presents the findings to stakeholders based on the information collected previously (styles, scenarios, utility tree, etc.).

These steps are grouped into two phases. The first phase (step 1 to step 6) is architecture-centric, in which information is gathered, elicited, and analyzed. The second phase is stakeholder-centric that consists of building a utility tree (based on the utility tree of phase 1), collecting architectural approaches information, prioritizing scenarios, and mapping this information and scenarios onto architecture. Moreover, stakeholders' viewpoints are elicited and the results of the first step are verified.

ATAM has been validated with several software systems and domains, such as the Battlefield Control System (BCS) and software-intensive ecosystems (Boxer & Kazman, 2017). In fact, it is considered the most applied evaluation method in both academic and industrial contexts.

4.3.1.5 Scenario based architecture reengineering (SBAR)

SBAR is a reengineering software architecture method that supports the assessment of multiple quality attributes. The evaluation consists principally of scenarios and performing design transformation to improve the targeted quality attributes. This process is performed iteratively until all requirements are met (figure 4.4). The inputs of this method are the software architecture and the updated requirement specifications, whereas the output is the improved design. SBAR consists of four main steps to perform assessment and transformations as follows (Bengtsson & Bosch, 1998).

a. *Incorporating new functional requirements*. In this stage, quality attributes are not addressed directly. Based on the updated specification, the decomposition of the system into its main components is redesigned at a high abstraction level and thus, the first architecture version is generated.

b. *Software quality assessment.* The goal of SBAR is to evaluate the potential of the designed architecture to achieve quality requirements. The method uses mainly scenario-based analysis to assess quality attributes Moreover; three complementary approaches can be used, which are simulation, experience-based reasoning, and mathematical modeling. Scenarios are particularly used to evaluate reusability and maintainability since they have proven their ability to express potential system changes. The evaluation through scenarios can be done in three sub-steps, which are defining a representative set of scenarios, analyzing architecture, and summarizing results. The simulation approach is used particularly to assess operational quality, such as fault-tolerance and performance. The main architecture's components are implemented whilst other components are simulated, which generates an executable system. The context, inside which the system is supposed to execute, is also simulated at an appropriate abstraction level. Mathematical modeling is more suitable for the static evaluation of the design models. It can serve as an alternative to simulation since both approaches can assess operational qualities. Experience-based approaches allow assessing quality attributes based on experience and logical reasoning based on that experience. In this context, the evaluation process is more subjective and less explicit since it is based on experience and intuition.

c. *Architecture transformation.* After the architecture evaluation has been completed, the assessed values are compared to the specified requirements. If unsatisfactory qualities are discovered, then the architecture must be changed to meet these qualities. The reengineering of the architecture consists of performing architecture transformations, each of which leads to generating a new architecture version that has the same functionalities but differs in its

quality attributes' values. There are many categories of transformation that can be applied to architecture, such as imposing specific architectural styles and patterns.

d. Software quality assessment. Steps 2 and 3 are repeated until all quality requirements are met.

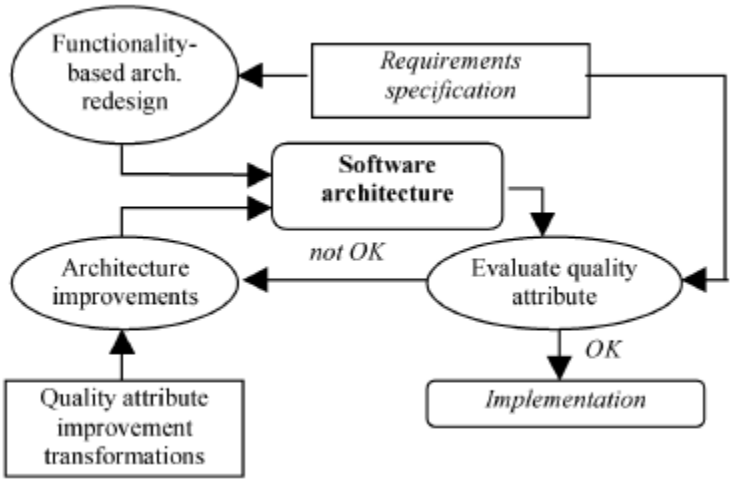

Figure 4.4. SBAR activities

4.3.1.6 Cost-Benefit Analysis Method (CBAM)

CBAM is an architecture-centric analysis approach destined for the economic modeling of software and systems. It was developed upon ATAM, in which the cost and benefits of architectural decisions are modeled to optimize such decisions. In this context, cost and benefits are traded off with system quality goals, and thus decisions are made in terms of these benefits as well as in terms of quality attribute response (Kazman et al., 2001).

Once ATAM finishes the analysis, CBAM starts based on the ATAM documented artifacts, such as the description of business goals, architectural views, risks, and tradeoff points. ATAM allows uncovering architectural decisions and links them to business goals and quality attribute response, whereas CBAM allows determining the cost and benefits associated with these decisions. Accordingly, quality attributes can be traded off based on the associated cost/benefits and the importance degree of each attribute, which may aid stakeholders in the decision-making process. CBAM consists of six activities as follows.

- *Choosing scenarios and architectural strategies*. Once the set of prioritized scenarios is generated by ATAM, a set of possible architectural decisions is described and associated with each scenario for which improvement is desired. The output of this step is a set of improvements, the affected portions of the existing architecture, and the description of architectural strategies.
- *Assessing quality attribute benefits*. The decision-making aiding process consists of determining cost and benefits. The authors consider determining cost as a well-established software engineering component; therefore, they focus principally on benefit. The latter is correlated with the support degree that architectural strategies can provide to quality goals,

which are related to business goals. It should be noted that both goals are outputs of ATAM. This information is used by CBAM to determine the importance of each involved quality attribute in the architecture. Each stakeholder assigns a score to each QA (e.g., performance: 25 and modifiability: 20 according to a 0:100 grading scale).

- *Quantifying the architectural strategies' benefits.* The assigned quality scores in the previous step are used to evaluate each architectural strategy. The latter can affect multiple quality attributes to varying degrees by contributing negatively or positively to their achievement. This can be captured by assigning -1 to the negative contribution and +1 to the positive one. Subsequently, the benefit of each architectural strategy is calculated by multiplying the corresponding quality score by the contribution value. The obtained results allow ranking the benefits of every considered architectural change. Furthermore, to capture uncertainty, CBAM uses stakeholder judgment variations for uncertainty measurement.
- *Quantifying the architectural strategies' cost and schedule implication.* In this step, the expected cost of each architectural strategy is estimated. Moreover, the schedule implication of each strategy is estimated in terms of critical shared resources, elapsed time, and dependencies among implementation efforts.
- *Calculating desirability.* In this step, a desirability metric is calculated. This metric can be used to rank architectural strategies and help in the decision-making process.
- *Making decisions.* In this step, architectural strategies are chosen according to cost, benefit, schedule implication, and desirability metric.

4.3.1.7 Architecture-Level Modifiability Analysis (ALMA)

ALMA is a scenario-based analysis method that focuses exclusively on *modifiability*, which means it does not support tradeoffs. It is characterized by its multiple analysis goals, repeatable techniques for performing analysis steps, and making implicit assumptions explicit. This method consists of the following steps (Bengtsson et al., 2004).

- *Goal setting.* This first activity consists of determining the analysis goals. These goals are maintenance cost prediction, risk assessment, and software architecture selection.
- *Architecture description.* In this step, information about the software architecture is collected. This information allows evaluating scenarios in two steps, which are analysis of the impact of scenarios and impact expression. ALMA depends on architectural views to provide information such as the decomposition of the system into components, the relationships between those components.
- *Change scenarios elicitation.* The goal of this step is to find and select the change scenarios required for the evaluation in the next step. However, the number of possible changes is probably infinite. ALMA combines two techniques to overcome this issue. The first technique (equivalence classes) consists of dividing scenarios into equivalence classes and selecting one scenario as representative for each class, which may reduce the number of scenarios significantly. The second technique (classification of change categories) uses top-

down or bottom-up approaches to classify the important change scenarios according to a selection criterion.

- *Change scenarios evaluation.* ALMA applies the architecture-level impact analysis to determine the impact of changes scenarios on software architecture. This type of analysis is divided into three steps: identifying the affected components, determining the effect on the components, and determining ripple effects. Subsequently, the results of the analysis can be expressed either quantitatively or qualitatively using quantitative measures (e.g., ++, +, -, --) to facilitate comparing the effects of scenarios.
- *Interpretation.* In this step, the obtained results are interpreted according to the system requirements and the analysis goal.

ALMA has been validated in several cases and domains, such as Althin medical, DFDS Fraktama, and software architectures at Ericsson Software Technology.

4.3.1.8 Active Reviews for Intermediate Design (ARID)

Active design reviews (ADR) are a technique proposed primarily to ensure quality by evaluating the detailed design of modules or components of a software system. Reviewers are asked to utilize the design and test actual understanding. ARID is a combination of ADR with a scenario-based architecture evaluation method, which is ATAM to tackle the problem of evaluating preliminary designs. This method is designed to perform in two main phases: pre-meeting and review meeting to assess one quality attribute namely *usability*. The first phase encompasses four steps (step1 to step 4), whereas the second phase encompasses five steps (step 5 to step 9) (Clements, 2000).

- Step 1: *Identifying reviewers*. A set of people who are supposed to present at the review is identified.
- Step 2: *Preparing design presentation*. The goal of this step is to prepare a presentation that explains in detail the design.
- Step 3: *Preparing seed scenarios*. The designer prepares a set of seed scenarios (like ATAM) to explain the concept of scenarios to reviewers. The latter can accept or refuse the provided scenarios.
- Step 4: *Preparing the review meeting*. In this step, the main review meeting is scheduled, reviewers are invited to attend the meeting, and copies of the review agenda, seed scenarios, and presentation are distributed to the invited reviewers.
- Step 5: *Presenting ARID method*. In this step, the meeting starts, and the ARID activities are explained to participants.
- Step 6: *Presenting design.* In this step, the design is presented to determine if it is usable. Participants can ask clarification questions (rationale or implementation questions are not allowed) to reveal potential issues that the designer must address before the design is considered complete.

- Step 7: *Brainstorming and prioritizing scenarios*. Similar to ATAM, participants suggest scenarios to solve potential problems that can encounter. The defined scenarios are put with seed scenarios into the pool. Subsequently, the voting starts by allowing each reviewer to select 30% of the total of the gathered scenarios. The scenarios that have the most voting percentage are used to test the usability of the design.
- Step 8: *Performing review*. Starting with scenarios that have received the most votes, reviewers are asked to produce codes that use the services provided by the design to resolve the problems stated by these scenarios. This step continues until the time fixed for the review has ended, all high-priority scenarios have been exercised, or the group indicates that a conclusion has been reached.
- Step 9: *Presenting the conclusion*. In the final step, a list of issues is presented and the participants are asked for giving their opinions concerning the review efficacy.

Unlike ATAM, ARID is not extensively validated in several projects and domains.

Discussion on scenarios-based methods

Scenario-based analysis and evaluation methods were addressed extensively in the literature. SAAM is considered the first one that has used scenarios to evaluate software architecture. All the subsequent methods have inspired this technique by SAAM explicitly or implicitly. Table 4.1 shows the main differences and similarities between the addressed evaluation methods according to the comparison criteria stated in the first column. The first criterion depicts that except SBAR and ARID, all methods perform the evaluation in the final version of SA at the design stage. In this context, there are three main phases of interest to architecture evaluation, namely early, medium, and post-deployment (Abowd et al., 1997). In the first two phases, the architecture is still immature and needs iterative elaborations. On the other hand, the architecture is considered mature (final version) in the third phase where the system has been completely designed, implemented, or deployed. The second criterion allows categorizing evaluation methods according to the number of quality attributes that can be assessed. Methods, such as ESAAMI, ARID, and ALMA are dedicated to evaluating one attribute, whereas methods like ATAM and SBAR are able to assess multiple attributes. The third criterion depicts that most of the addressed methods do not support tradeoffs among quality attributes, except for ATAM and ARID. The fourth criterion showed us the most of the evaluation methods support the involvement of all stakeholders in the evaluation process. The fifth criterion illustrates the number of activities performed by each method during architecture evaluation. ATAM and ARID encompass the highest number of evaluation activities performed in two phases (nine activities in two phases), whereas SAAMER comprises the lowest number of activities (four activities).

The main problems encountered with the majority of scenario-based methods are a) the results and effectiveness of analysis depend on the selection, elicitation, and relevance of scenarios

that can identify and cover the most critical assumptions and defects within SA, b) there is no fixed minimum set of scenarios that makes the analysis meaningful except for some methods that defined techniques to stopping scenarios generation such as SAAMER, and c) the lack of tool support.

It should be noted that the list of scenario-based methods addressed in this dissertation is not meant to be exhaustive. Actually, a long list of these methods can be found in (Babar et al., 2004; Dobrica & Niemela, 2002; Sobhy et al., 2021). This reflects the richness of this evaluation category with diverse methods and techniques in particular, and the complexity of the domain of software architecture evaluation in general.

Scenario-based methods inspired us to define the first service of MS-QuAAF that allows analyzing and evaluating the architecture at the design stage to discover defects before going any further in the development process. However, the evaluation within our framework is performed quantitatively through metrics in order to provide assessors with more accurate evaluation results. Furthermore, the analysis through this service allows checking the adherence of the designed architecture against the specified one and thus fixing irregularities as early as possible.

Method comparison	**SAAM**	**ESAAMI**	**SAAMER**	**ATAM**	**SBAR**	**CBAM**	**ALMA**	**ARID**
Applicable project stage	Final version of SA	Final version of SA	Final version of SA	Final version of SA/ integrated into an improvement process during design	Reengineering stage	After ATAM analysis	Final version of SA	During components design
Quality attribute	Mainly modifiability	modifiability	Reusability and evolution	Multiple attributes	Multiple attributes	Cost and benefit	modifiability	usability
Tradeoffs	Not considered	Not considered	Not considered	considered	considered	considered	Not considered	considered
Stakeholders involvement	All	All	All	All	designers	All	Architects	designers
# evaluation activities	6 activities	6 activities	4 activities	9 activities in 2 phases	4 activities	6 activities	5 activities	9 activities in 2 phases

Table 4.1. Scenario-based architecture evaluation methods

4.3.2 Goal decomposition analysis methods

In the field of software engineering research, there are three types of goal decomposition analysis methods, namely the Fault Tree Analysis (FTA), the Goal Structuring Notation (GSN), and the Softgoal Interdependence Graph (SIG). The FTA (Ruijters & Stoelinga, 2015) is a top-down deductive analysis method that decomposes the root fault into subsequent causes using a tree structure form. The GSN (Kelly & Weaver, 2004) is a graphical notation used to visualize thoughts and methods supposed to lead to the achievement of stakeholders' objectives by decomposing the top goal into sub-goals supported by evidence. The SIG is used within the NFR framework (Chung et al., 2000) to represent the NFR refinement structure, defining top quality requirements, and determining whether the system can achieve those requirements. In this chapter, we will focus on the NFR framework and SIG-based methods since they are dedicated to quality evaluation and analysis in a quality assurance context. Furthermore, MS-QuAAF defines a new technique derived from SIG to evaluate architecture at the implementation stage.

4.3.2.1 The NFR framework

The NFR framework is a qualitative goal-oriented approach that uses non-functional requirements (performance, security, etc.) as a means to drive the overall design process. The framework depends principally on Softgoal Interdependence Graphs (SIGs) to represent and record the reasoning and design process. A softgoal is *a goal (an NFR) that has no clear-cut definition and/or criteria as to whether it is satisfied or not* (Chung et al., 2000). A SIG is a graphical representation that illustrates the interdependencies between softgoals using lines or arrows (figure 4.5). More specifically, SIGs are constructed by following a set of major steps:

- *Identifying NFRs.* The main NFRs that the system should achieve are identified. Figure 4.5 shows two main NFRs, namely good performance for account and secure account. Each NFR is represented as a top softgoal or the root node in the corresponding SIG.
- *Decomposing NFR softgoal.* Each main softgoal is divided into a set of sub-goals. For instance, the security node is broken into integrity, confidentiality, and availability of accounts. The decomposition is performed iteratively until no further decomposition is possible.
- *Dealing with priorities.* Softgoals considered of high importance are annotated with the symbol "!".
- *Identifying possible operationalizations.* Operationalizations represent design alternatives that can be implemented as development techniques (functions, data, constraints, etc.) to achieve NFR softgoals. Operationalizations softgoals are drawn as thick clouds, and they represent another type of softgoals. For instance, the *Use indexing* cloud (node) is an operationalization or a development technique that can be implemented to enhance the *Response time* NFR softgoal.

- *Dealing with implicit interdependencies among softgoals.* Operationalization softgoals can contribute positively (+) to one NFR softgoal and negatively (-) to another. The relationships between these two types of softgoals are represented by dashed lines for implicit interdependencies (correlations), and solid lines for the explicit ones. For instance, using uncompressed format may contribute positively to response time and negatively to storage space. Consequently, the tradeoff time-space is detected with the help of correlations.
- *Selecting among alternatives.* The refinement process continues until no other alternatives need to be taken into consideration. The resulted SIGs are now completed and available to assist developers in selecting adequate alternatives using the information provided by those SIGs (operationalizations, tradeoffs, etc.).

Plenty of methods have extended the NFR framework and its SIG to evaluate software architecture qualitatively or quantitatively in different academic and industrial contexts.

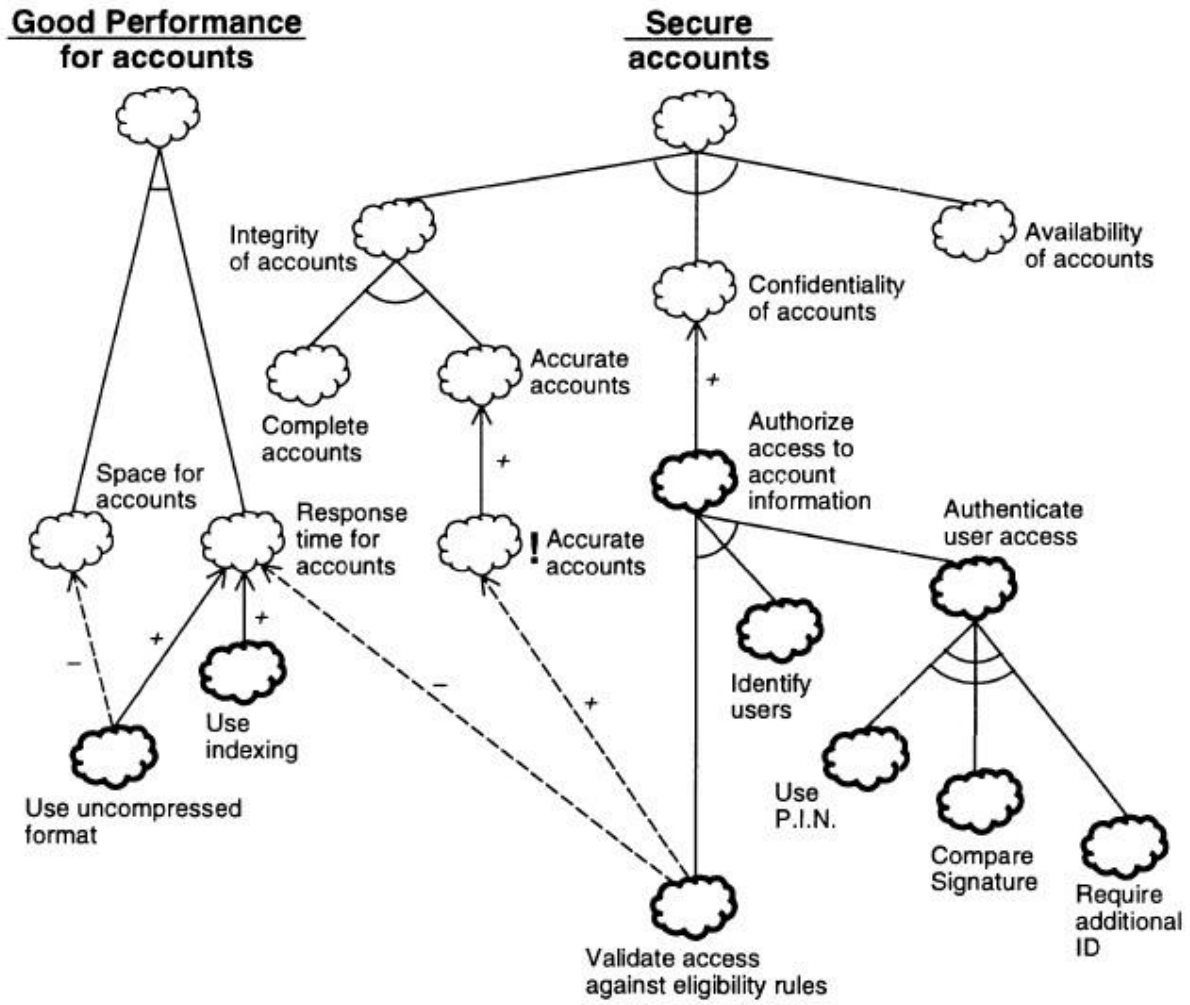

Figure 4.5. An example of performance and security SIGs

4.3.2.2 A goal-oriented simulation approach for cloud-based system architecture

This approach codenamed Silverlining (Chung et al., 2013) is a combination of goal-oriented requirements engineering with cloud computing simulations to capture stakeholders' goals in

private cloud-based systems. These goals are used to explore, analyze, and select among architectural alternatives at early stages to provide profitable, performant, and scalable cloud-based architecture. The simulation is used to investigate the impact of infrastructural design choices on the ability of the system to meet stakeholders' needs. This approach necessitates 6 steps to output the design that is probably more suitable for these stakeholders taking into consideration conflicting goals.

- Step 1: *Goals identification.* Stakeholders' goals are discovered, analyzed, and refined.
- Step 2: *Conflict identification*. The goals identified in the previous step are represented using an Agent-SIG diagram. The latter differs a little bit from the original SIG by representing the relationships between stakeholders (agents), their goals, and their roles in the system. Although this representation is clearer, refinements are required to decompose those goals into specific operationalizations, which are assigned to the corresponding agents.
- Step 3: *Enriching the qualitative goal model with quantitative information*. The Agent-SIG diagram is augmented with quantitative domain-specific information to support performing simulations in order to assess the impact of architectural decisions on the softgoals satisfaction degree. Stakeholders' requirements from the application domain are translated into numerical values to which the eventual design is expected to meet in order to satisfy the related softgoals. These values are called design constraints, which can be used throughout the requirements elicitation for cloud-based projects. The presented method used these constraints to focus on three target goals, namely profitability, scalability, and performance.
- Step 4: *Iterative simulation and model refinement*. The constraints obtained in the previous step are translated into simulation models using a dedicated simulation tool called CloudSim. The obtained models allow investigating the impact of design decisions on the softgoals presented by the Agent-SIG diagram. These models are refined by adding new architectural decisions based on the initial simulation results and the unmet goals to improve the captured design toward meeting stakeholders' goals.
- Step 5: *Translating the derived model into the system architecture*. The obtained models from the simulation process are translated into the cloud-based architecture in the specific application domain. Although the simulation can reduce the number of architectural alternatives, the design space may remain large, which makes exploring all designs infeasible. Based on architects' experience, the optimal design with respect to tradeoffs is selected and improved subsequently.
- Step 6: *Testing*. In order to minimize the gap between the ideal world of simulators and the application real world, a set of tests must be performed. These tests are dedicated principally to reducing the cost by detecting failures before implementing the architecture and purchasing hardware.

4.3.2.3 A quantitative assessment method for analyzing safety and security using the NFR approach

This method (Subramanian & Zalewski, 2014) applies the NFR approach to assess quantitatively safety and security in Cyber-physical systems (CPSs). Firstly, the authors applied the qualitative evaluation to a CPS using the NFR approach. Second, the qualitative aspects of the produced SIG are mapped to quantitative aspects according to a set of mapping rules (M1, M2, etc.) as follows (figure 4.6).

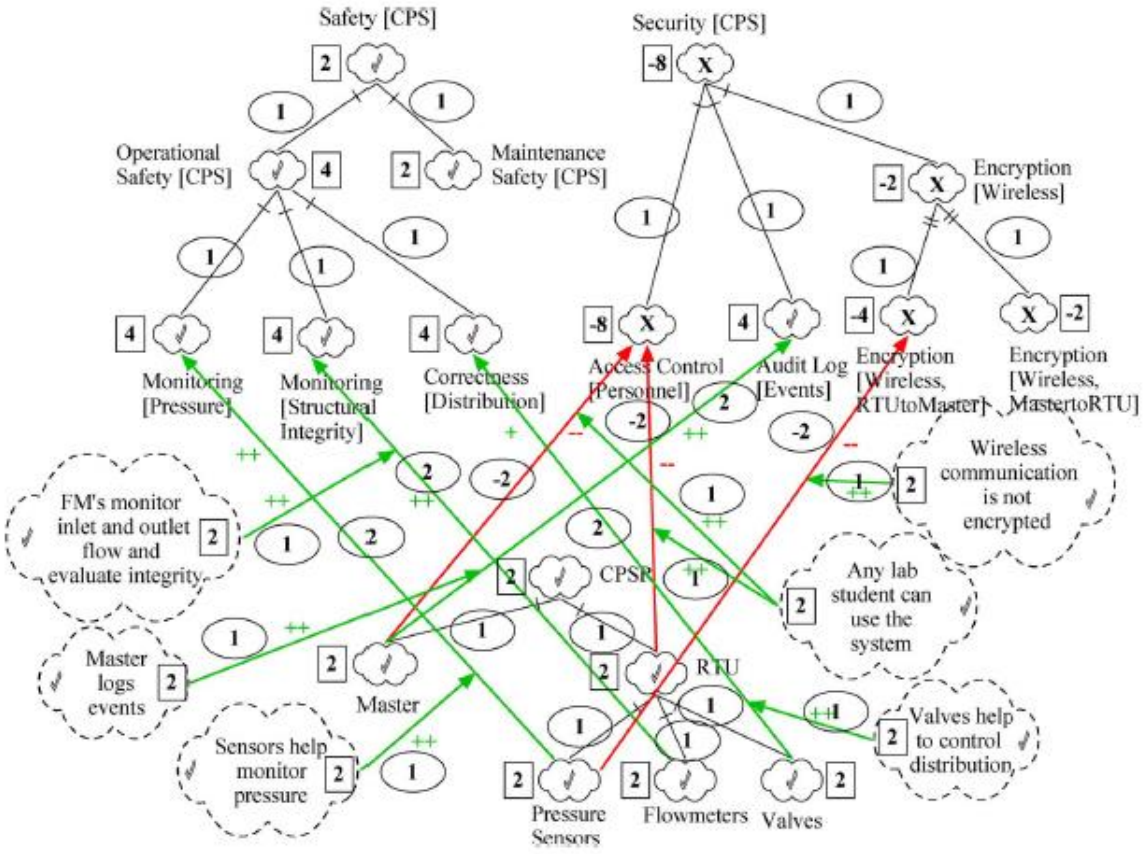

Figure 4.6. A quantitative SIG using the SV mapping scheme

M1. Leaf softgoals are converted into a corresponding metric. Therefore, satisfied or denied leaves whether they are operationalizations or NFR softgoals are converted into the appropriate metrics.

M2. Contributions (MAKE, HELP, HURT, and BREAK) are converted to the corresponding quantitative representation.

M3. If criticalities are associated with contributions or softgoals, then criticality metrics are generated accordingly.

M4. The quadruple (leaf softgoal metric, leaf softgoal criticality metric, contribution metric, and contribution criticality metric) is transformed into a metric that represents the individual contribution of a leaf softgoal to its direct parent.

M5. The metric of the parent softgoal is calculated from the collection of all the individual contributions of the child softgoal.
M6. Similarly, the metric of the parent contribution is calculated from the collection of all the individual contributions of the child softgoal.

The obtained metrics are propagated recursively through a bottom-up process, in which each evaluated softgoal is considered as the next leaf softgoal. The authors followed the single-value (SV) scheme as the quantitative scheme to apply the above mapping rules. However, they claimed that other schemes could be applied, such as fuzzy logic or probabilistic. This evaluation method has applied the SV scheme as follows.

- M1 provides five values, which are 2, 1, -1, -2, and 0 to represent satisfied, weakly satisfied, weakly denied, denied, and unknown softgoals, respectively.
- M2 attributes the scores +1, +0.5, -0.5, and -1 for the contributions MAKE, HELP, HURT, and BREAK, respectively.
- M3 provides the criticalities values: 0 for a non-critical softgoal, 1 for a critical softgoal, 0 for a non-critical contribution, and 1 for a critical contribution.
- M4 provides a formula dedicated to combining the metrics for the above-mentioned quadruple.
- M5 gives a formula to calculate the metric of the parent softgoal.
- M6 provides a formula to calculate the metric of the parent contribution.

The results of the case study shown in figure 4.6 reveal that the top softgoals safety and security have obtained scores 2, and -8, respectively, which means that safety is better than security. This is due to the obtained metrics of contribution and NFR softgoals during the evaluation process.

4.3.2.4 A quantitative architecture evaluation using the goal decomposition framework and Archimate

This method proposes a quantitative goal decomposition tree to clarify and assess the NFR goals that the system should promote (Zhou et al., 2020). It supports the evaluation of several quality attributes, unlike the above-mentioned approaches. The NFR goal tree is inspired by the NFR framework, in which the evaluation is performed qualitatively. The tree is constructed by decomposing recursively the top claim (softgoal or goal) into a set of sub-claims. To each leaf sub-goal, an evidence node is attached. The latter (close to the concept of operationalization) may represent a design specification, a test, a requirement definition, or a system operation. A modeling tool called Archimate is used to construct this tree and visualize the system at the early stages of the development process. More specifically, the NFR goal tree is created by applying the following main steps in sequence (figure 4.7).

Step1: *Goal identification, decomposition, and prioritization.* First, the most important NFR goals are identified and structured. Second, the top NFR goal is decomposed recursively into several sub-goals until no further decomposition is possible. Third, priorities (weights) are

defined among sub-goals of the same parent. For instance, the sub-goals B and C have the same priority, which is ½.

Step2: *Attribute values assignment*. First, evidence (yellow nodes) is attached to each leaf node. Second, attribute values are assigned to each evidence according to an adjustable quantitative scale (2: strongly satisfied, 1: satisfied, 0: neutral, etc.).

Step3: *Attribute value calculation.* In this step, the top claim is calculated by propagating values from the bottommost nodes. First, values are propagated from evidence nodes to sub-goals according to one-to-one relationship or N-to one relationship. Second, values are propagated recursively from sub-goals to their parent node (P) according to the values of the sub-goals (S1, S2, etc.) and their weights (W) using the following equation. $\sum i=1, k\ Si*Wi$, where $\sum i=1, k\ Wi= 1$. The obtained result allows determining whether a top claim is satisfied and can be achieved through the target architecture.

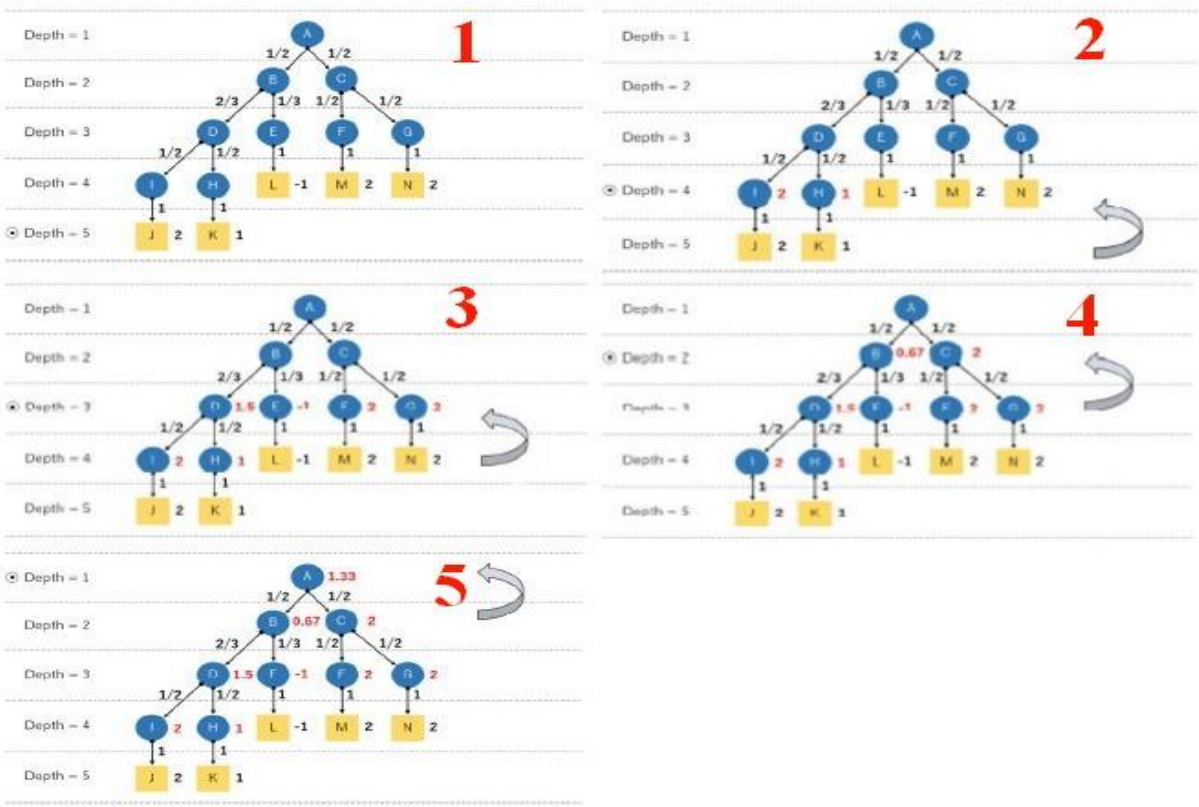

Figure 4.7. Quantitative architecture evaluation using the NFR goal tree

Discussion on goal decomposition analysis methods

In this sub-section, we have presented a subset of methods that use the goal decomposition technique in general and the NFR framework in particular to evaluate software architecture. This subset is not intended as an exhaustive list. There are several qualitative and quantitative methods proposed in the literature, such as (Kaiya et al., 2002; Kokune et al., 2007; Saito & Yamamoto, 2006; Yamamoto, 2015).

Table 4.2 illustrates the main differences and similarities between the aforementioned methods. The first method (the NFR framework) proposed by Chung provides a new idea and perspective on how to evaluate conflicting quality attributes (NFR goals). The idea consists mainly of constructing and using a Softgoal Interdependency Graph to choose among design alternatives, determining the achievement of the NFR goals, and resolving the conflict between the competing goals. Several methods have extended the NFR framework and used the concept of SIG to evaluate architecture qualitatively, quantitatively, or a combination of both. The second method combines a qualitative SIG with a cloud simulation tool to choose among design alternatives that may satisfy the stakeholders' goals in private cloud-based systems. This method supports the evaluation of three NFR goals, which are profitability, performance, and scalability. The authors did not mention whether their approach could be applied to other NFR goals within other development contexts. Moreover, this approach seems long and needs several simulation iterations alongside cloud development expertise to choose among the most suitable alternatives. The third method is dedicated to assessing safety and security and resolving the conflicts between them. The evaluation consists of mapping qualitative aspects of the constructed SIG into quantitative aspects, calculating metrics, and propagating the obtained metrics to the parent nodes recursively. This method requires two steps to perform the mapping process, whereas other methods, such as (Yamamoto, 2015) can apply directly the quantitative evaluation without mapping, which may decrease the evaluation time and effort. Besides, this method cannot be applied to architectures where safety is not considered a system priority. The fourth method is a quantitative evaluation method that supports the assessment of multiple NFR goals. This method uses weights to determine the priority among sub-goals on the one side, and increasing the accuracy of the final achievement score on the other side. Unlike the NFR framework, this method does not support tradeoffs among NFR goals. Additionally, according to its authors, it is not clear how to apply the evaluation in large-scale systems where a high number of interdependencies are established between goals and sub-goals.

In summary, the goal decomposition analysis has provided a new way to evaluate NFR goals that are not clear-cut. However, there are some shortcomings that may affect the evaluation accuracy. First, qualitative methods lack statistical significance, and thus cannot give us exactly the degree of satisfaction of the NFR goals. Second, quantitative methods may need several iterations to calculate the satisfaction of each goal. Besides the decomposition, weightage, values attribution, and values propagation steps, deeper goal trees, and large SIGs may lengthen the evaluation process due to the increased number of iterations. Moreover, the exactitude of the obtained results depends on the accuracy of the scores assigned to leaf nodes at the beginning of the evaluation process. Third, it is difficult to apply this type of evaluation to systems where goals are not easily mapped to a tree or graph structure. Fourth, most of the methods that we found in the literature are dedicated narrowly to assessing security and safety; therefore, a lack of multi-attribute methods that can assess a large set of NFR goals is addressed.

MS-QuAAF proposes a new technique derived from the NFR framework to evaluate architecture at the implementation stage. This service proposes a quantitative evaluation tree called the Responsibilities Satisfaction Tree (RST) inspired by weighted SIG and goal decomposition tree, in which nodes represent NFR responsibilities (this term will be explained in chapters five and six) instead of softgoals. RST attempts to improve the accuracy of evaluation by proposing dedicated equations to calculate the weight of nodes and their relationships contrarily to the above-mentioned techniques where weights are estimated by designers and architects.

Method / **Comparison**	**The NFR framework**	**A goal-oriented simulation approach for cloud-based system architecture**	**Quantitative assessment method for analyzing safety and security using the NFR approach**	**Quantitative architecture evaluation using the goal decomposition framework and Archimate**
Development stage	Design stage	Design stage	Design stage	Design and requirement analysis
Quality attribute	Multi-attribute	Profitability, performance, and scalability in private could-based systems	Safety and security	Multi-attribute
Tradeoff	Supported	Not supported	supported	Not supported
# Evaluation activities	6 activities	6 activities	6 activities	3 main activities

Table 4.2. A comparison between the presented goal decomposition analysis methods

4.3.3 Metric-based evaluation methods (metrics suites)

In software engineering, metrics are proposed as methods and tools to quantify attributes in software projects, products, and processes (Fenton & Bieman, 2019; Sommerville, 2011). The main motivation for proposing measurement is to obtain quantitative values that reflect the status of the software quality. Once these values are collected, they are compared to other standardized values defined across an organization. The comparison allows drawing conclusions about the software quality and thus determining where it stands against the stakeholders' objectives.

Starting from the middle of the 1970s, many design metrics were proposed to assess specifically software internal characteristics, such as size and complexity. These traditional metrics are considered generic because they are not destined for a particular design or programming paradigm. However, after the object-oriented design has become the new trend and the dominant paradigm that most organizations shifted to, most of the proposed metrics suites were destined for evaluating architectures that adopted this approach of design. These metrics were defined to assess specific object-oriented properties, such as coupling between objects,

inheritance, and so forth. In this sub-section, we will only discuss the traditional and object-oriented metrics due to their relevance to software architecture and quality attributes (figure 4.8). On the other hand, metrics that do not measure the system design are considered out of the scope, such as software project metrics (processes, tests, etc.).

Despite the design paradigm, quality attributes targeted by metrics can be divided into two main categories, external and internal. The former represents the visible and the perceived properties that the user can experience, such as usability and reliability. Generally, these properties are related to the dynamic behavior of the system and they can be measured directly or indirectly after the product has been created. The latter represents the static properties of the software that the end-user never sees. They are related to the internal design where the execution of the system is not required. Modularity, lines of code, and degree of reuse are examples of internal attributes. Measuring these attributes allows not only understanding the system quality but also predicting or measuring indirectly the external attributes. This implies that if internal properties are altered, then the external ones may also be affected in the process.

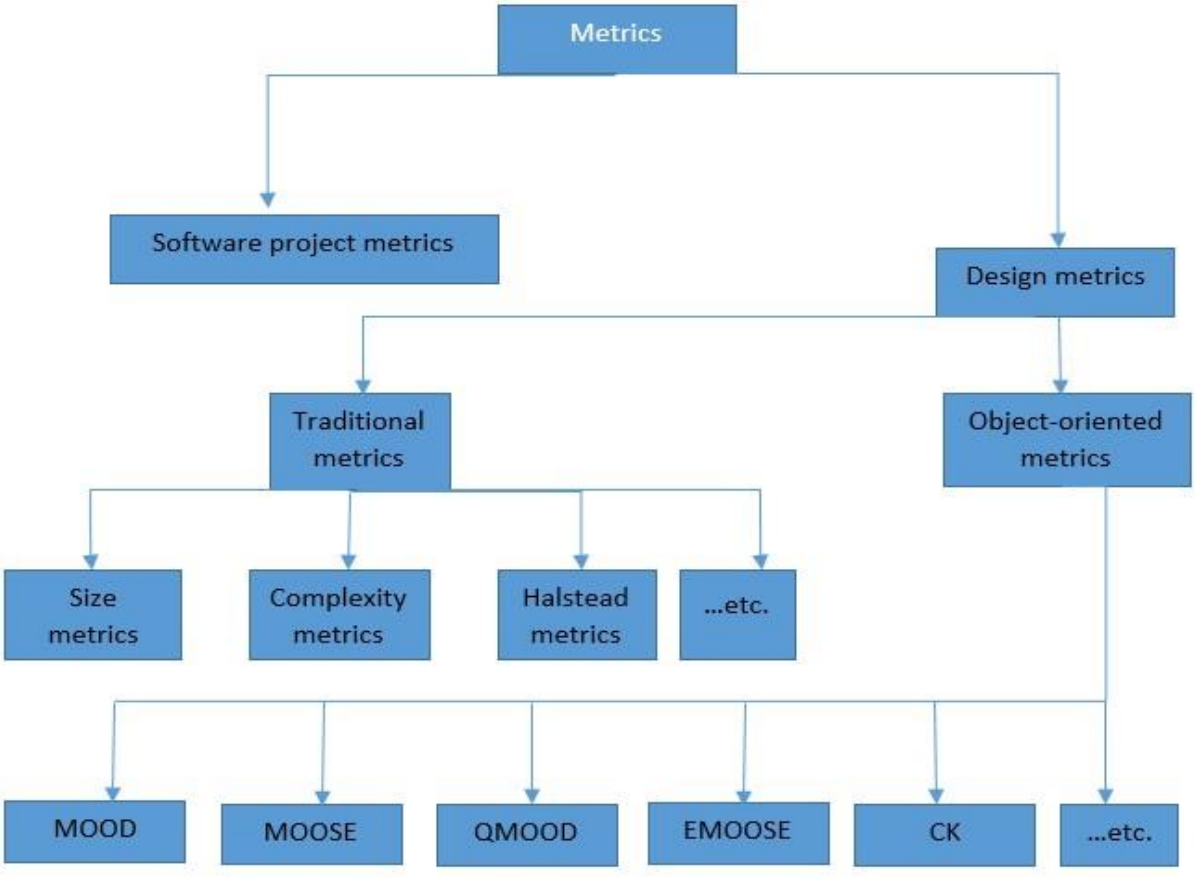

Figure 4.8. A subset of design-based metrics

4.3.3.1 Traditional metrics

This sub-section discusses traditional metrics that have been proposed mainly to measure some essential design and programming properties, such as size and complexity.

4.3.3.1.1 Size related metrics

This type of metric is dedicated to estimating and predicting the development effort by quantifying the size of the software. The following metrics are the most known ones in this category.

- *Source lines of code (SLOC)*. Known also as LOC, this metric is used to measure the size of software by counting the number of lines of its source code (Lorenz & Kidd, 1994). This metric is defined to predict programming productivity, development effort, and maintainability. However, there is no standard to define what a line of code is, or what are program entities must be considered when calculating the size of software (comments, declaration, statement, etc.).
- *Function Point Metric (FPM)*. This metric is proposed to measure the work effort at the early development stages. This effort is predicted by measuring the number of functions that the software should perform (Albrecht & Gaffney, 1983). A function is quantified as function points, which are inputs, outputs, master files, and inquiries. The authors have proposed a two-step procedure to estimate effort. First, function points are used to estimate SLOC. Second, SLOC is used to estimate the development effort.
- *Bang*. This metric is proposed to measure the size of a software product during the requirement analysis based on the components of the structured analysis description (DeMarco, 1986). The measurement consists of dividing the system into three categories: function-strong, data-strong, and hybrid systems. The classification is based on the ratio RE/FP, where RE is the number of relationships in a retained model, and FP is the number of functional primitives in a data flow diagram (DFD). If this ratio is lesser than 0.7, then the system is function-strong. If the ratio is greater than 1.5, then the system is data-strong. Otherwise, the system is hybrid. For function strong systems, a function bang (function metric) is calculated by identifying a set of primitives from the design diagram, and then the number of tokens of each primitive is calculated using dedicated equations. For data-strong and hybrid systems, the number of objects is identified, and then the number of relationships of each object is calculated.

4.3.3.1.2 Complexity metrics

This type of metric is dedicated primarily to quantifying the complexity of software in order to predict the development effort.

- *McCabe's Cyclomatic Complexity (CC).* This metric was proposed to measure the complexity of software programs (source code) using control flow graphs (McCabe, 1976). The graph's nodes represent indivisible commands of a program connected by edges (control flows). The Cyclomatic complexity represents the maximum number of the linearly independent paths within a program's source code. In this context, an independent path has at least one edge that has never been traversed by other paths. The formula proposed to calculate this complexity metric is the following.

V (G) = E – N + 2P, where E is the number of edges, N is the number of nodes, and P is the number of predicate nodes (condition nodes). The larger the number of paths, the higher the program complexity. Furthermore, V (G) is equivalent to the maximum number of test cases that could be performed to test a software program. This metric has been correlated with several quality attributes, such as maintainability, modularity, and understandability.

- *Halstead's software metrics.* According to Halstead, a computer program is a set of tokens classified either as operators or as operands. Halstead has proposed a set of metrics (10 metrics) that count these tokens in order to measure complexity and estimate the development effort (Halstead, 1977). To do that, he has defined a set of basic measures as follows.

  n1: the number of distinct operators.

  n2: the number of distinct operands.

  N1: total number of occurrences of operators.

  N2: total number of occurrences of operands.

  n*1: the minimum possible number of operators

  n*2: the minimum possible number of operands.

  By using these measures, some of the proposed metrics were computed as follows.

  - The length of a program is N=N1+N2.
  - The vocabulary of a program is n=n1+n2.
  - The volume of a program is $V=N*\log_2 n$.

These metrics were integrated into many tools to measure complexity and programming effort.

4.3.3.2 Object-oriented metrics

Since the object-oriented design had gained huge popularity in the software development community, several metrics suites have been proposed to assess the quality of architectures that follow particularly this design paradigm. These metrics are intended for evaluating object-oriented design properties, and thus judging the software architecture accordingly.

4.3.3.2.1 Metrics for Object-Oriented Software Engineering (MOOSE)

MOOSE (or CK) is the most-known object-oriented measurement suite. The latter was proposed by Chidamber and Kemerer to assess OO design (Chidamber & Kemerer, 1994). They have defined six metrics to check the consistency and the integrity of the design against a set of OO characteristics, such as coupling between objects, inheritance, and cohesion. More specifically, they have correlated with each metric the following significance and roles.

- *Weighted Methods per Class (WMC*). This metric is related directly to complexity, which is calculated by summing the complexity of all methods defined within a class. WMC can be used to predict the time and effort needed to develop and maintain a class. A large number of methods of a class will affect its children since they inherit all the methods, which may increase significantly the complexity of those children classes. Furthermore, a high number of methods means that the class is more application-specific, which decreases its reusability.

- *Depth of Inheritance Tree (DIT).* The depth of inheritance of a class is the maximum length from the root of the inheritance tree to the node representing this class. DIT allows measuring how the number of ancestors can affect a class. The larger the value of DIT, the deeper the class in the inheritance hierarchy and the greater the number of inherited methods, which makes predicting the behavior of this class more complex. Additionally, deeper trees imply complex design since more classes and their methods are involved. However, a high DIT value may indicate also a greater possibility of reuse, unlike WMC.
- *Number of children (NOC*). This metric gives us the number of the direct subclasses of a class in the class inheritance hierarchy. The higher the value of NOC, the greater the reuse. Moreover, a greater NOC may imply improper parent class abstraction and sub-classing misuse.
- *Coupling Between Object Classes (CBO).* In the OO design, two classes are said coupled if the methods of one class use methods or variables declared in the other class. In this context, the CBO of a class is the number of classes to which it is coupled. A higher CBO may prevent reusability and lessen modularity. In addition, maintainability becomes harder since the sensitivity to changes in other related parts is greater. Therefore, keeping the CBO at the minimum allows designing more independent classes, and thus improving modularity and maintainability.
- *Response for a class (RFC).* The response for a class is the cardinality of the set of methods that can be called and executed in response to a message sent to an object of this class. This metric can measure the potential communication between classes. The larger the value of RFC, the higher the complexity of the class and the harder its debugging and testing.
- *Lack of cohesion in methods (LCOM)*. Cohesion has been always used to measure how elements within a module or a package are strongly related. The CK suite measures the cohesion inside a class by the similarity between pairs of methods. Two methods are considered similar if they use the same variables. LCOM is computed by subtracting the count of pairs whose similarity is not zero from the count of pairs whose similarity is zero. If the number of non-similar methods is high, then the class is less cohesive and vice versa.

4.3.3.2.2 Chen's object-oriented metrics

Chen has proposed a set of metrics to measure specifically the complexity of a class within an object-oriented design. He has defined the following metrics to measure coupling, cohesion, complexity of operations, and so forth.

- *Operation complexity metric*. This metric was proposed to measure the operation (method) complexity of a class. It is computed by summing the complexity value of all the operations defined inside a class.
- *Operation argument complexity metric*. This metric is calculated by summing the complexity value of each argument declared in each operation of a class.

- *Attribute complexity metric*. This metric is computed by summing the complexity value of each attribute defined in a class.
- *Operation coupling metric*. This metric measures the coupling between the operations of a class and the operations defined in other classes. It can be calculated by summing the number of operations' accesses to/by other classes.
- *Class coupling metric*. This metric is dedicated to measuring the coupling between classes. It is calculated by summing the number of accesses to/by other classes.
- *Cohesion metric*. This metric assesses the cohesion of a class by measuring the number of intersections between the arguments of operations. It is calculated by dividing the number of disjoints set resulted from the intersection of sets of arguments by the number of these sets.
- *Class hierarchy metric*. This metric is calculated by summing the depth of a class in the inheritance tree, the number of its sub-classes, the number of its super-classes, and the number of inherited or local operations available to this class.
- *Reuse metric*. This metric is equal to 1 if the class is reused, otherwise, the metric is equal to 0.

4.3.3.2.3 Metrics for Object Oriented Design (MOOD)

Abreu has defined a metric suite called MOOD to evaluate the main concepts of the object-oriented paradigm. MOOD measures particularly the concept: encapsulation, information hiding, inheritance, polymorphism, message passing, and reuse. The main goal of MOOD is to promote reusability and maintainability by setting design recommendations. The metrics proposed within MOOD are the following.

- *Attribute Hiding Factor (AHF)*. This metric is used to measure the concept of encapsulation (information hiding). A high AHF is recommended because it means that most attributes are hidden (only accessed by the corresponding methods).
- *Method Hiding Factor (MHF)*. This metric is also used to measure encapsulation. The number of visible methods is an indicator of the class functionality. A low MHF reflects increased functionality; however, it indicates a weak abstraction. On the other hand, a high MHF points out low functionality.
- *Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF).* These two metrics are used to measure inheritance. A higher number of inheritance relationships indicates a deeper and wider inheritance tree, which may affect testability and understandability negatively.
- *Polymorphism Factor (PF).* This metric allows assessing polymorphism by measuring the ratio of possible polymorphic situations to the maximum number of possible polymorphic situations.

- *Coupling factor (COF).* This metric measures the coupling degree between classes. A high COF value indicates high coupling and thus increased complexity, which is not recommended OO design.

4.3.3.2.4 A Hierarchical Quality Model for Object-Oriented Design (QMOOD)

Bansiya has proposed a hierarchical assessment model for high-level design quality attributes at the early stages of the development process (Bansiya & Davis, 2002). This model proposes a suite of metrics to assess behavioral and structural object-oriented properties, such as inheritance, modularity, and encapsulation. These properties are correlated with high-level quality attributes, such as complexity, flexibility, and reusability. The relationships from OO properties to quality attributes are weighted according to their importance in the system design. More specifically, the following eleven metrics were proposed to measure eleven OO properties in order to assess six quality attributes (reusability, flexibility, understandability, functionality, extendibility, and effectiveness).

- *Design Size in Classes (DSC).* This metric designates the total number of classes.
- *Number of Hierarchies (NOH).* This metric denotes the number of class hierarchies.
- *Average Number of Ancestors (ANA).* The value of this metric represents the average number of ancestors along all paths of a root class.
- *Data Access Metric (DAM).* This metric denotes the ratio of private attributes to the total number of attributes defined in a class. The higher the value of DAM, the better is the design.
- *Direct Class Coupling (DCC).* This metric signifies the number of direct relationships created between classes through attribute declarations and message passing.
- *Cohesion Among Methods of Class (CAM).* This metric indicates the ratio of the sum of the intersections of parameters of a method to the maximum number of parameters in the class.
- *Measure of Aggregation (MOA).* This metric signifies the number of attributes declaration whose types are classes defined by the user.
- *Measure of Functional Abstraction (MFA).* This metric designates the ratio of methods inherited by a class to the number of methods accessible by member methods of that class.
- *Number of Polymorphic Methods (NOP).* The value of this metric is the number of polymorphic methods in a class.
- *Class Interface Size (CIS).* This metric represents the number of public methods in a class.
- *Number of Methods (NOM).* The value of this metric signifies the number of all methods declared in a class.

A tool called QMOOD++ was developed to collect data from C++ projects and compute these metrics automatically.

4.3.3.2.5 Li's object-oriented metric suite

Li proposed a new set of metrics claimed to be an alternative suite to the CK metrics. The new suite was proposed to remedy deficiencies identified during the validation of the CK suite using the Kitechenham's metric-evaluation framework (Kitchenham et al., 1995).

- *Number of Ancestor Classes (NAC).* Using the inheritance tree, this metric measures the count of ancestor classes from which a class inherits. It is the same as the CK's DIT.
- *Number of Descendent classes (NDC).* This metric measures the count of sub-classes inherited from a class in the inheritance tree. It is equivalent to the CK's NOC.
- *Number of local methods (NLM).* This metric measures the total number of public (accessible outside a class) local methods defined within a class. The higher the value of NLM, the more effort is needed to comprehend, implement, test, and maintain the class.
- *Class Method Complexity (CMC).* This metric is calculated by summing the complexity of all local methods (public and private) defined in a class. Similar to NLM, a high CMC may affect understandability, implementation, testing, and maintenance.
- *Coupling Through Abstract Data Type (CTA).* This metric is calculated by summing all classes that represent abstract data types of attributes declared in a class.
- *Coupling Through Message Passing (CTM).* This metric represents the number of messages sent from a class to other classes except messages sent to local objects in the local methods of this class. The higher the values of CTM and CTA, the more effort is needed to perform implementation, testing, and maintenance.

4.3.3.2.6 Choi's Component-based metrics

Component-based metrics have not widely addressed in the literature since component-based architectures are based on object-oriented technology, in which components generally consist of a set of related classes. Therefore, the above-mentioned metrics can be also used to assess component-based architectures. In this context, properties like coupling and cohesion can be assessed by measuring the intra-relationships between classes of the same component or inter-relationships between classes scattered over different components. Choi's (Choi et al., 2009) applied the strength of dependency between classes (SDC) to define the metrics: Cohesion of a Component (CHC), Coupling of a Component (CPC), Coupling Between Components (CBC), Average Cohesion of a System (ACHS), Average Coupling of a System (ACPS), and Independence Degree of a System (IDS=ACHS-ACPS). These metrics measure coupling and cohesion at the component level, between components, and globally at the overall system level.

Discussion on metric-based methods

Quality metrics were proposed to provide stakeholders with statistical information about the software system. This information represents measurement values related mostly to the internal characteristics of the system. Table 4.3 illustrates the comparison between traditional metrics and OO metrics suites. Traditional metrics have focused primarily on size and complexity

characteristics to estimate the development effort and predict maintainability. These metrics can be applied to any source code despite its programming paradigm. However, most of them do not allow measuring structural and behavioral design properties, which narrows their usefulness to the size and complexity of source code at advanced stages of the software lifecycle. On the other hand, object-oriented metric suites were proposed specifically to assess object-oriented designs. They allow measuring OO internal design properties such as coupling, encapsulation, and inheritance to draw conclusions about external characteristics. In this context, maintainability, testability, reusability, and modularity are the most correlated external attributes to OO internal properties. Although OO suites have widened the extent of quantitative evaluation using metrics through exploring and assessing deeper design properties, they cannot be applied to design and programming paradigms outside the OO approach.

Metrics-based methods can suffer from many drawbacks. First, most metrics are used to assess design properties after the architecture has been implemented. This implies that potential defects are discovered at late stages of the development process; therefore, adjusting those defects can be very costly in terms of time, money, and development effort. Second, many metric suites lack tools that can collect design data from source code and compute metrics automatically to improve and expedite the evaluation process. Third, the presented metrics focus mainly on assessing structural properties, which diminish the number of assessed quality attributes to a small set. Fourth, concepts, such as size, complexity, coupling, and cohesion are often misunderstood and have inconsistent definitions in the literature (Briand et al., 1996).

Taking into consideration these drawbacks, we have proposed MS-QuAAF, a metric-based framework that defines a set of metrics to assess software architecture. The framework allows evaluation at both design and implementation stages. Moreover, the defined metrics are generic, which allows assessing any inputted attribute on the condition that the architectural decisions taken to promote these attributes are specified at the early development stages.

Metrics / **Comparison**	**Traditional metrics**	**Object-Oriented metrics**
Development stage	Implementation stage	Mostly implementation stage
Design/programming paradigm	general	Object-Oriented
Main objectives	Estimating the development effort	Assessing reusability, maintainability, testability and modularity
Tools	Size and complexity metrics have been integrated into many tools	Few metric suites have developed the corresponding evaluation tool.

Table 4.3. A comparison between traditional and OO metrics

4.3.4 A myriad of other approaches to assess and predict quality attributes

There are countless of other quantitative methods that propose different techniques to predict software architecture. However, we found that most of these methods focus principally on two quality attributes, either maintainability or performance. Koziolek categorized in his survey (Koziolek, 2010) five main approaches to predict performance in component-based software, which are UML-based approach, meta-models-based approach, middleware-based approach, formal performance specification approach, and measurement-based approach. The first four methods predict performance at design time, whereas the fifth one measures performance after the system has been implemented. Concerning maintainability, it is considered the most addressed quality attribute in the literature. Most of the proposed methods have attempted to assess maintainability, either quantitatively or qualitatively. In addition to the above-mentioned methods (scenario-based, OO methods, etc.), at least five categories of approaches were proposed to evaluate maintainability (Malhotra & Chug, 2016), which are statistical algorithms (e.g., Markov hidden models), machine learning algorithms (neural network, artificial neural network, etc.), nature-inspired techniques (e.g., genetic programming), expert judgment, and hybrid techniques (e.g., genetic algorithms with neural networks). These categories are out of this dissertation's scope and will not be depicted in the next chapters.

4.4 A concluding discussion about the presented evaluation methods

In this chapter, we have presented three categories of evaluation approaches (table 4.4) addressed widely by several researchers in the literature, especially scenario-based and metric-based approaches. These approaches are somewhat similar to MS-QuAAF in many aspects, particularly architecture analysis, architecture evaluation, and quality attributes. We have defined six main criteria (section 4.2) to classify those approaches, which are measurability, evaluation technique, development stage, number of quality attributes, tradeoffs, and tool support. These criteria allow comparing different evaluation techniques as well as putting emphasis on their advantages and liabilities as follows.

- *Measurability*. Although qualitative methods have the ability to assess a large set of quality attributes compared to the quantitative ones, they suffer from subjectivity and the lack of quantitative data that can reflect the quality achievement more accurately.
- *Evaluation techniques*. Each evaluation technique has its specificity, advantages, and drawbacks. Scenario-based methods were proposed to assist architects in requirement elicitation, design alternatives comparison, and gauging the architecture against anticipated or expected use of the system. However, scenario-based methods may suffer from other problems besides their qualitative nature, such as the relevance of scenarios against the most critical assumptions, and the number of defined scenarios that makes the analysis meaningful. Goal decomposition methods have provided a new way to assess NFR goals that are not clear-cut by using goal decomposition analysis. However, this technique cannot be applied to systems where goals are not easily mapped to a tree structure form. Besides,

the exactitude of evaluation results depends on the experience degree of evaluators since they have a crucial role in assessing goals satisfaction in the first steps of the evaluation process. Metric-based methods provide assessors with quantitative and statistical data that reflect the state of some internal characteristics of architecture, which can help to judge external attributes. Nevertheless, concepts such as complexity and size have inconsistent definitions, which may lead to different interpretations of the same architecture. Furthermore, these methods focus narrowly on assessing complexity at the implementation stage, which limits the number of assessed quality attributes to those correlated to complexity.

- *Development stage*. Architecture analysis at design time has many benefits, such as detecting irregularities at early stages and reducing maintenance cost. However, this type of analysis does not allow continuing the evaluation at advanced stages, such as implementation; therefore, evaluating the realized architecture is not supported. On the other hand, the analysis at the implementation stage allows extracting from the source code useful information, such as coupling and cohesion. However, it is very costly to perform changes at this stage as we have mentioned before.
- *Number of quality attributes.* Some quality attributes are proven hard to quantify, such as security and reliability. As a result, quantitative methods are generally proposed to assess specific quality attributes, such as performance and maintainability. However, quite a few quantitative goal decomposition methods have proposed a multi-attribute evaluation. On the other hand, qualitative methods can support a larger set of attributes, though few scenario-based methods support multiple attributes evaluation, such as ATAM and SBAR.
- *Tradeoffs.* Considering tradeoffs among quality attributes can provide more accurate assessment results. We have found that OO metrics do not consider tradeoffs during the evaluation process. On the other hand, the number of scenario-based and goal decomposition methods that support tradeoffs is somewhat equal to methods that do not support tradeoffs.
- *Tool support.* Most of the presented methods, either quantitative or qualitative do not provide tool support that can facilitate and automate the evaluation process.

The analysis of these methods and the comparison between them gave us a clear idea on the domain of software architecture evaluation, although its diversity and complexity. Accordingly, we extracted from these evaluation techniques the main strengths and weaknesses to develop a new framework that encompasses new features within a new generic evaluation perspective.

Criteria \ Approach	Scenario-based	Goal decomposition framework	Metric-based
Measurability	Mostly qualitative	Qualitative, quantitative, and hybrid	quantitative
Evaluation technique	Through scenarios	Based on the NFR framework	Measurement through metrics
Development stage	Design stage	Design stage	Mostly implementation stage
# quality attributes	Fewest methods support multi-attribute evaluation	Most of methods focus on safety and security	Maintainability, testability, reusability, and modularity
Tradeoffs	Not all methods consider tradeoffs	Not all methods consider tradeoffs	Not considered
Tool support	Almost all methods do not provide tool support	Almost all methods do not provide tool support	Most methods do not provide tool support

Table 4.4. A comparison between all the presented evaluation methods

4.5 Conclusion

The analysis of the evaluation methods presented in this chapter allowed us to discover the following main drawbacks.

- The evaluation is performed, either at the design time or implementation stage. Therefore, there is no such feature as continuous evaluation.
- There is no conformance analysis between the prescribed architecture and the descriptive one. As a result, architecture deviation analysis is not included within these methods
- Most methods support the evaluation of one or two quality attributes, especially quantitative methods.
- Qualitative methods lack statistical significance.
- Metric-based methods assess only some internal characteristics (especially complexity) correlated with a small set of quality attributes.
- Lack of tool support.

These shortcomings are the rationale for proposing MS-QuAAF, a new generic metric-based framework for evaluating software architecture continuously at the design and implementation stages. The framework attempts to overcome many of the drawbacks stated above by integrating

three services within the same framework to a) providing continuous evaluation, b) supporting multi-attributes evaluation, and c) assessing architecture defects and conformance. The details of the evaluation process using MS-QuAAF will be explained in the three next chapters.

Chapter 5

The framework MS-QuAAF for monitoring and evaluating architecture quality: foundation and evaluation methodology

In the first part of this dissertation, which includes chapter 2, chapter 3, and chapter 4, we have presented the context within which our work and evaluation approach will be carried out. In the second part, which commences with the current chapter, we will depict in detail the evaluation framework MS-QuAAF.

In this chapter, we will present firstly the rationale behind proposing the evaluation methodology within MS-QuAAF. Second, the framework's methodology and its foundation will be presented in a nutshell. Third, we will depict the concept of architecture facets, model projection, and facet projection.

Contents

5.1 Motivations and rationale behind MS-QuAAF 94
5.2 An introduction to the MS-QuAAF's evaluation methodology 96
5.3 The facet projector module 101
5.4 Conclusion 112

5.1 Motivations and rationale behind MS-QuAAF

Software architecture is generally designed based on architectural decisions made at the early stages of the development process to achieve functional and non-functional requirements (*NFRs*) (Falessi et al., 2011). However, the ability of a software system to fulfill its assigned functional responsibilities does not imply that the target quality attributes are met. For instance, a system can deliver efficiently correct results but does not satisfy the security requirement by allowing malicious users to access easily its confidential data. This implies that the architectural decisions made are either violated or not suitable for delivering software of high quality. Therefore, architecture must be evaluated and monitored continuously throughout the whole development process and at each maintenance activity. It should be an innovative and powerful methodology to do that, yet simple and easy to implement, which is our primary focus in this work.

In the previous chapter, we have studied and presented the most well-known evaluation approaches and techniques in order to investigate their abilities and weaknesses. Based on this study, we have fixed clearly our evaluation methodology's goals and designed the skeleton of MS-QuAAF (Kadri et al., 2021b) accordingly. Subsequently, we have built, tested, and integrated gradually each assessment service until the framework has been completely developed.

The presented evaluations approaches were divided mainly into two basic categories, namely qualitative and quantitative. The former uses questioning techniques to generate qualitative questions to be asked of an architecture whilst the latter suggests quantitative measurement to evaluate an architecture. Although qualitative evaluation is thought to be applicable to assess any given architecture quality (Abowd et al., 1997; Bass et al., 2012), it lacks statistical significance. Questioning techniques like questionnaires and checklists are mostly based on the evaluators' perspective and subjectivity, which may decrease the evaluation accuracy and objectivity. Additionally, the results of scenario-based analysis depend on the selection of the scenarios and their relevance for identifying architecture's weaknesses. In this context, there is no fixed number of scenarios to guarantee that the evaluation analysis is meaningful (Dobrica & Niemela, 2002). On the other hand, the essence of measuring techniques is to deliver assessors with quantitative results and views that can reflect the state of the architecture quality more accurately. However, as we have stated previously, this type of evaluation can suffer from the following issues:

- Quantitative evaluation frameworks are addressed to answer specific questions, and thus evaluate specific qualities, such as performance (Koziolek, 2010) and modifiability (Riaz et al., 2009). This is due to the fact that some quality attributes are hard to quantify.
- Defining metrics for some attributes, such as security, and usability have proven difficult to develop (Bachmann et al., 2005).
- Many metrics are used to evaluate the architecture at one specific development stage, more specifically, at design time or after the product is complete or nearly complete (Bansiya & Davis, 2002). Therefore, a lack of evaluation and risk-management frameworks that can cover these main stages throughout the development process (design time, implementation, and deployment) can be addressed.

With the purpose of overcoming the main shortcomings of quantitative and qualitative techniques, we have proposed a quantitative evaluation framework (belongs to the metric-based category) called **MS-QuAAF** (**M**ulti-**S**ervice - **Qu**antitative **A**rchitecture **A**ssessment **F**ramework). The framework defines a suite of generic evaluation metrics to help evaluators in:

- Assessing any quality attribute inputted into the framework.

- Assessing continuously architecture throughout two main stages of the development process (design and implementation). Two architecture states will be examined, which are the designed architecture and the implemented one.
- Estimating architecture defectiveness and detecting potential deviations using dedicated metrics.
- Evaluating the achievement of the NFR responsibilities (see chapter 6) assigned to promote quality attributes using a dedicated metric and through the goal decomposition tree (see chapter 4).
- Making decisions about the architecture, for instance, architects can approve or disapprove the design (or parts of it) based on the obtained evaluation results.

MS-QuAAF is a multi-service assessment framework derived from ISO/IEC/IEEE 42030:2019 ("ISO/IEC/IEEE International Standard - Software, Systems and Enterprise – Architecture Evaluation Framework," 2019), the generic architecture evaluation framework that specifies objectives (quality attributes), approaches, factors, and evaluation results as the key elements of any instantiated framework. MS-QuAAF performs architecture evaluation through two main modules. The first module proposes the concept of facet projection to extract from architecture's meta-models only information of interest to the evaluation task. The second module proposes seven metrics applied to the target architectures through three assessment services to evaluate these architectures at the design and implementation stages. These metrics are called generic because they can measure the satisfaction of any quality attribute inputted into the framework. The latter does not provide specific metrics for maintainability, or performance, or any other quality attribute. In contrast, the proposed metrics are common to measure all targeted quality attributes on the condition that the architectural decisions taken to promote these attributes are specified at early development stages. At the design stage, metrics are used to judge the correctness of the designed architecture against the established architecture specification. At the implementation stage, metrics are used to judge the fulfillment of the responsibilities assigned to promote quality attributes. The evaluation results at each stage allow architects to accept or refuse the deviation from architecture specification. However, rectifying architecture irregularities at the design time is somewhat easier and less costly compared to the implementation stage where performing major changes is burdensome, complex, and very costly in terms of time and money.

5.2 An introduction to the MS-QuAAF's evaluation methodology

Architecture evaluation can be defined by the judgments about architectures according to the specified evaluation objectives ("ISO/IEC/IEEE International Standard - Software, Systems and Enterprise – Architecture Evaluation Framework," 2019). MS-QuAAF judges software architecture by analyzing architecture specification with respect to stakeholders' non-functional requirements (NFR) goals. An architecture specification encapsulates all architectural decisions

taken to satisfy these goals, such as architectural styles, tactics, and design rules (constraints). These architectural decisions constitute meta-models to which the designed and implemented architecture must comply.

The essence of the evaluation adopted by the framework is based on the ATAM's (Kazman et al., 2000) idea that claims that architectural styles and tactics are the main determiners of quality attributes (chapter 4). Therefore, architectural decisions are evaluated continuously to judge stakeholders' quality attributes. In this context, the continuous evaluation provided by the framework consists of:

- Analyzing and assessing the architecture at the design stage.
- Analyzing and assessing the architecture at the implementation stage.
- Keeping on providing evaluation services at each major maintenance iteration.

The rationale for providing such evaluation is that most of the frameworks assess the architecture at one development stage, which does not allow quality monitoring and improvement throughout the key stages of the development process. More specifically, design time frameworks allow only early evaluation of the architecture to determine the extent to which architectural decisions meet the quality requirements. However, they do not permit architecture evaluation at late stages, such as implementation and deployment. On the other hand, the evaluation at late stages permits capturing the dynamic behavior of architectural decisions (Sobhy et al., 2021), detecting architecture erosion (de Silva & Balasubramaniam, 2012), and verifying the achievement of specific stakeholders' concerns; however, it is not designated for assessing architecture at high abstraction levels (structures, topology, styles, constraints, etc.).

Contrarily, MS-QuAAF allows quality monitoring throughout two main stages of the development process, more specifically, after the accomplishment of the design and implementation stages. The former evaluation allows identifying defects, which helps architects to fix design flaws according to the early architecture specification. The latter assesses the satisfaction of the NFR responsibilities prescribed to promote stakeholders' quality attributes. As a result, incorporating these two types of assessment within the same framework allows:

a) Continuous architecture evaluation and monitoring.
b) Identifying if a poor architecture quality is caused by rules infringement at the design stage, implementation stage, or both of them.
c) Improving the quality by adjusting the architecture in accordance with the architecture specification.

Additionally, the evaluation is performed quantitatively through a set of metrics to provide architects and evaluators with quantitative views that reflect the state of architecture quality more accurately.

Conceptually, the framework MS-QuAAF is organized around two basic modules as follows (Figure 5.1).

- **The facet projector**: This module is charged with analyzing architecture specifications to project out (extract) only elements of interest designated to promote the quality attributes under assessment. These elements constitute what we call the *architecture facet*. This module will be described extensively in the current chapter, in which the concepts of facets, model projection, and facet projection will be explained thoroughly.
- **The quality evaluator**: This module represents the core of the proposed framework. It is charged with evaluating architecture or parts of architecture using a set of assessment services (table 6.1). Firstly, it reads the architecture facet produced by the first module as input. Secondly, according to the development stage, it starts the corresponding service to apply the appropriate metrics. Finally, a conclusive assessment report is produced as output. This module will be depicted in the next chapter.

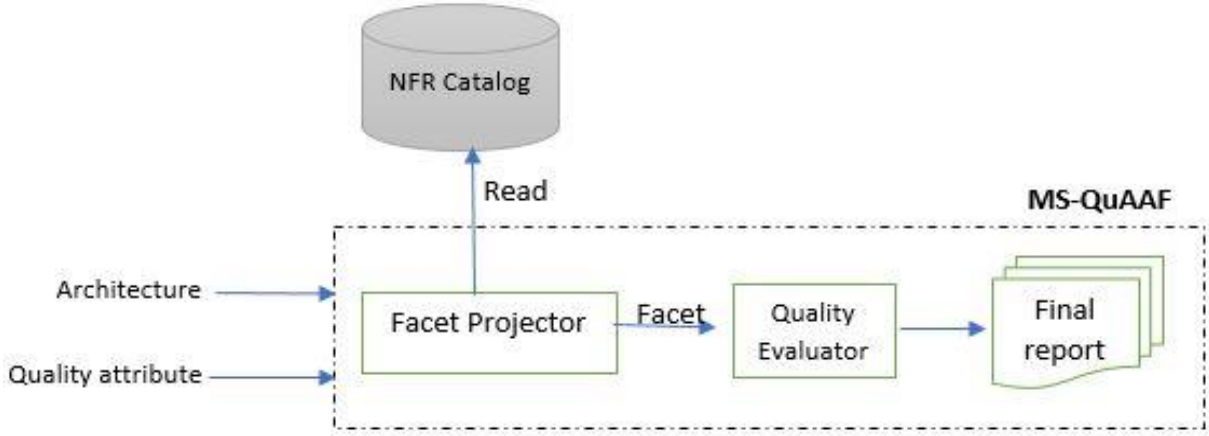

Figure 5.1. A simplified conceptual model of MS-QuAAF

5.2.1 The ISO/IEC/IEEE 42030:2019 architecture evaluation framework

MS-QuAAF follows the standard ISO/IEC/IEEE 42030:2019 ("ISO/IEC/IEEE International Standard - Software, Systems and Enterprise – Architecture Evaluation Framework," 2019), which is a specification of a generic architecture evaluation framework for software, systems, and enterprises. The specification consists of a set of evaluation sub-clauses, practices, and principles according to which derived or specific frameworks must adhere. The major aims of this standard are to enable and assist architecture evaluation in order to

a) Validating architectures supposed to promote stakeholders' quality attributes.
b) Assessing architecture quality regarding the fixed objectives.
c) Assessing progress in contrast with architecture objectives achievement.
d) Identifying potential architectural risks.
e) Supporting decision-making where architecture is involved.

f) Determining whether architecture entities satisfy their intended purposes.

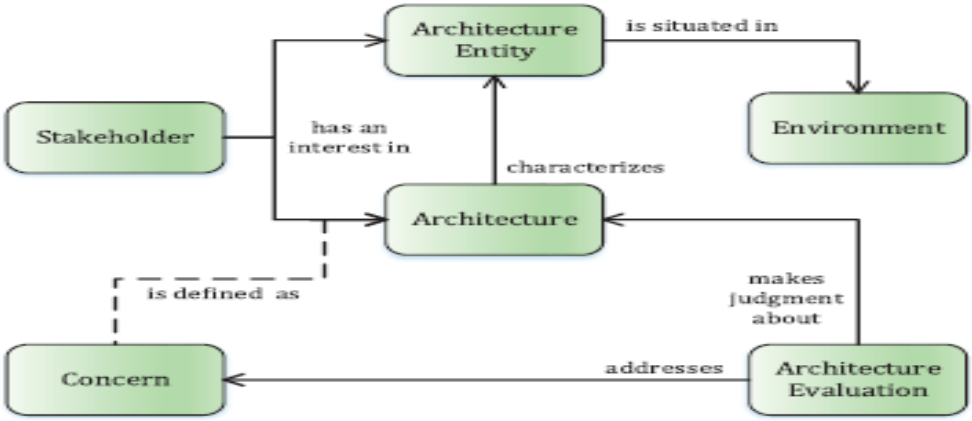

Figure 5.2. The architecture evaluation context

Figure 5.2 shows the context within which the evaluation effort can be performed to address stakeholders' concerns defined through architecture. The evaluation allows determining the potential of an architecture entity by making judgments against the extent to which concerns or objectives have been achieved. Therefore, these concerns are the primary focus of architecture evaluation frameworks.

At a high abstraction level, this standard specifies a set of sub-clauses according to which every Architecture Evaluation (AE) framework should include the following key elements.

a) At least one AE objective: An objective depicts the extent to which stakeholders' concerns and the related business drivers are addressed.
b) At least one AE approach: An approach is the means to specify AE objectives.
c) At least one AE factor: a factor can contribute to one or more objectives. Cost and schedule are examples of AE factors.
d) At least one AE result: A result is an outcome of using an AE approach.

5.2.2 The compliance of MS-QuAAF with the ISO/IEC/IEEE 42030:2019

The standard does not provide approaches that depict how to assess an architecture entity. In fact, it defines only a set of sub-clauses according to which specific derived frameworks should adhere. In this context, MS-QuAAF follows the above sub-clauses; however, it proposes its own evaluation approaches performed through a set of three services to provide stakeholders with evaluation reports at the key development stages. More specifically, MS-QuAAF complies with these sub-clauses as follows.

a) *AE objectives*: An objective represents a stakeholder concern, which is a quality attribute in the context of this framework. The latter is designated to assess any quality attribute if and only if the architecture specification contains the corresponding architectural decisions.
b) *AE approaches*: MS-QuAAF is a multi-service and multi-step evaluation framework, in which each service follows an evaluation approach that applies its defined metrics.

These approaches constitute the evaluation methodology defined by the framework to provide a continuous architecture evaluation. In the design stage, the dedicated service analyzes and assesses the defectiveness of architectural rules, styles, and tactics specified in the early stages. The defectiveness metrics allow judging the correctness of the designed architecture against the established specification by analyzing rules infringements. For instance, the closer the value of defectiveness is to zero, the better the quality of the architecture. Therefore, the quality is judged by the framework based on the density of architecture defects caused by rules violations. In the implementation stage, the dedicated service allows checking the satisfaction of architectural rules after the accomplishment of the implementation stage. For this purpose, the service uses a responsibilities decomposition technique to calculate the satisfaction indicator. The latter is compared with optimal, moderate, and dissatisfactory calculated values to judge the architecture in this stage. In the final evaluation stage, a concluding evaluation report is generated by the corresponding service. The latter uses an evaluation matrix that contains the results obtained by the previous services to make final judgments on architecture quality. Moreover, these services can be recalled to reevaluate the architecture at each major maintenance iteration.

c) *AE factors*: Factors are circumstances or facts that influence the evaluation process and contribute causally to its results, such as cost, performance, schedule, etc. For instance, allocating a sufficient evaluation budget allows recruiting more experienced architects, which may increase the evaluation performance and accuracy. On the other hand, low budgets, inexperienced architects, or unorganized evaluation schedules may affect negatively the evaluation outcomes and accuracy.

d) *AE results*: Each service produces an evaluation sub-report, which contains assessment values. Additionally, the framework is designed to produce the final report after the evaluation has been completed. At each stage, architects and evaluators judge the architecture by comparing the obtained results with the optimal and dissatisfactory results.

Although MS-QuAAF respects the standard's key evaluation sub-clauses, it is a customized framework that differs from the standardized evaluation framework in the following points:

- The standard states that the existence of architecture specification is not obligatory to perform an evaluation. Contrarily, MS-QuAAF considers the existence of architecture specification crucial because the evaluation effort performed within the framework is based on assessing the designed and implemented architecture against the specified architecture documentation.
- The standard states that architecture analysis is optional since information can be obtained from other sources (the standard did not specify which sources can be used).

Contrarily, MS-QuAAF depends on architecture analysis to detect defects and irregularities.

- The standard defines a generic evaluation framework for assessing software, systems, enterprises, and so forth. On the other hand, MS-QuAAF is dedicated exclusively to assessing software architecture quantitatively through a set of metrics.

5.3 The facet projector module

Software architectures are so complex, which means that describing them in a one-dimensional fashion is impossible (Bass et al., 2012). This implies that software architectures are not flat but rather multi-dimensional entities that encompass multiple facets and views. The tremendous complexity can make the architecture evaluation even harder and trickier. Therefore, we should expand our view to handle the architecture from multiple sides and angles according to stakeholders' quality attributes. Each side can reveal parts of the architecture. We call these sides the architecture facets (**AF)**. Accordingly, we believe that depicting software architecture as facets can enhance understandability significantly and thus streamlining quality assessment and monitoring. In this dissertation, AF is the first-class artifact used by MS-QuAAF to slice architecture through the facet projector module. The latter reads the architecture specification and the slicing criterion (which is the quality attribute under assessment) as input, and produces the architecture facet as output. The obtained facet will be sent to the quality evaluator module to be assessed by the designated services and metrics.

5.3.1 Architecture facet

Hiding information that is not necessary for the task to be undertaken is the essence of AF. It exposes only elements of interest mapped to the quality attribute under assessment to satisfy a specific evaluation task. More concretely, an AF encompasses only the architectural decisions made (topology, tactics, constraints, etc.) to promote this attribute. The latter is judged based on the results obtained from analyzing and evaluating the facet using the framework's metrics, which makes the evaluation much easier by assessing only the architectural decisions of interest. A facet is extracted from large meta-models that reside in the NFR catalog using facet projection (section 5.3.5).

Facets are mapped directly to quality attributes, for instance, a facet mapped to the portability attribute can expose the adopted architectural style (e.g., the layered style), the architectural elements in each layer, the relations among layers, and the tactics and rules specified to promote portability. Thus, a facet has the ability to depicting certain aspects of the architecture and ignoring the irrelevant ones, which characterizes its strength when we deal with large architectures.

5.3.2 The NFR catalog

In this dissertation, we prefer to use a new concept that we call the NFR catalog instead of NFR documentation. A non-functional requirement catalog is a repository that encapsulates architecture's design information and rationale. More specifically, it contains meta-models that

encompass all architectural decisions taken to satisfy stakeholders' NFR goals. These meta-models are specified generally through modeling and specification languages, such as OCL (Richters & Gogolla, 2002), Alloy (Jackson, 2012), Acme (Garlan et al., 2010), and so forth. Among the information entries contained in this catalog, we can categorize the following:

- The NFR responsibilities assigned to promote the desired quality attributes.
- The architectural decisions taken to fulfill NFR responsibilities, such as the adopted architectural styles, tactics, rules, constraints, and so forth.
- The involved architectural elements, their interfaces, properties, and the relations among these elements.
- The mapping between each quality attribute and the architectural decisions made.

The specification of this information constitutes large meta-models from which MS-QuAAF extracts facets to proceed with the evaluation of the related quality attributes.

MS-QuAAF depends heavily on the design information categorized in this catalog to extract facets, analyzing architecture, and assessing quality attributes. By interrogating the NFR catalog to extract facets, we can find that different facets promote different quality attributes and goals at different priority degrees. Quality attributes that represent the most concern to stakeholders are mapped to facets that contain design information of high design priority. In this regard, the higher the facet's importance is, the more its defectiveness is costly. Therefore, violating design rules that reside behind a high prioritized facet may have a harmful impact and catastrophic consequences on the overall architecture, and thus on its quality.

5.3.3 The anatomy of an architecture facet

The major purpose of facets is to hide the unnecessary information and show to us only the elements of interest to the evaluation task at hand. This information can be hidden or shown through the concept of views. A view is a representation of a set of system elements and the relationships associated with them (Clements et al., 2003). Anatomically, a facet is constructed by weaving two views, an architectural view, and a quality view (figure 5.3) (Kadri et al., 2020, 2021a):

- *The architectural view*: it is a structural view since it shows a set of structures and their pathways of interaction that architects have chosen to satisfy a specific NFR goal. What we get from this view is the result of the architectural decisions taken to fulfill the quality attribute depicted in the associated quality view. It should be noted that architectural decisions represent a blend of architectural styles, tactics, rules, and constraints combined to achieve stakeholders' goals.
- *The quality view*: this view shows principally the quality attribute to be achieved by the architectural decisions depicted in the architectural view. A quality attribute can be

external or internal. External attributes represent the qualities perceived by the end-user, such as security and performance. On the other hand, internal attributes represent the unperceived qualities by the end-user, such as reusability and modularity. However, despite the type of these attributes, their achievement is crucial from the stakeholders' point of view. This quality view shows as well other important entities, such as sub-attributes and the involved stakeholders.

These two views are weaved to construct the requested facet. We came up with this illustrative representation to boost up the understandability of the projected facets and thus improving the effectiveness of their usefulness. By using this view-based representation, MS-QuAAF can deduce directly the association between the adopted design and the relevant quality attribute. In other words, each facet exposes a set of structural elements and architectural decisions through its architectural view according to the quality attribute (and its sub-attributes) depicted in the quality view. For instance, the framework can proceed with the evaluation of the security attribute by assessing the associated architectural view that exposes the relevant architectural components, the adopted architectural styles, the assigned security responsibilities, and the architectural measures taken to provide security.

More formally, we describe a facet AF_i with the triplet $<AV_i, QV_i, R>$, where:

- AV_i is the architectural view of the facet.
- QV_i is the quality view of the facet.
- R is the relation that attaches AV_i to QV_i.

We describe an architectural view AV_i with the triplet $< A, E, D>$, where:

- $A== \{A_i /i=1, 2…n\}$ is the set of the architectural decisions taken to promote the quality attribute Qa_i depicted in QV_i.
- $E= \{E_i /i=1, 2…m\}$ is the set of the architectural elements (components, connectors, etc.).
- $D= \{D_i /i=1, 2…k\}$ is the dependencies among E, constrained by rules, tactics, and architectural styles.

We describe a quality view QV_i with the couplet $< Qa_i, Sh>$, where:

- Qa_i is the quality attribute promoted by architectural decisions depicted in AV_i.

- Sh= {Sh_i / i=1, 2 …m} is the set of stakeholders interested of the quality attribute Qa_i.

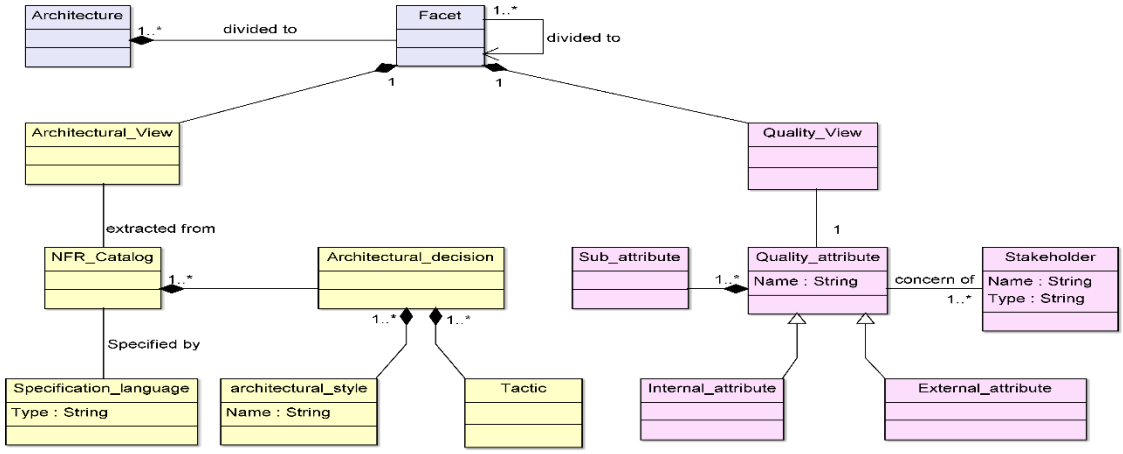

Figure 5.3. The anatomy of an architecture facet

The structure and representation of a facet are instantiated from the multi-view modeling paradigm presented in (Kadri et al., 2020, 2021a), in which meta-models and models are depicted as two interweaved views (architectural and quality views). In these two papers, we have proposed through the multi-view paradigm to model a software architecture at three abstraction levels (Kadri et al., 2019), which are the macro-level (components and connectors), the meso-level (classes and their relationships), and the micro-level (objects). In the process of developing MS-QuAAF, we have chosen the multi-view paradigm to represent facets because it provides an illustrative representation that allows deducing easily the relationships between the architectural decisions taken and the relevant quality attributes. Therefore, the reasoning about the architecture and the evaluation task can be enhanced, which may facilitate the quality evaluation significantly.

5.3.4 Facets Vs AOSD's aspects

Aspect-oriented software development allows the separation of concerns by modularizing crosscutting concerns into separate entities called the aspects (Pérez et al., 2006). AOSD permits both separate design and implementation of aspects, and their integration with other components of the system (Clemente et al., 2011). By this means, facets and aspects are almost dissimilar, and each one of them is addressed to resolve distinct problems. More specifically, aspects are designed and specified in the early stages of software design to be implemented in the subsequent coding stages (France et al., 2004; Stein et al., 2002). Besides, aspects aim to improve some quality attributes, such as understandability, adaptability, reusability, and maintainability (Laddad, 2003; Stein et al., 2002). On the other hand, facets are extracted from the architecture documentation and not designed or specified in the design phase. Additionally, facets represent on-demand artifacts used to assess quality attributes and not to improve them. In fact, the only concept shared between aspects and facets is modularization. The former uses modularization to encompass crosscutting concerns in separate modules. The latter uses

modularization to encompass design information related to one quality attribute. The reason behind this comparison is to avoid misinterpretations and clear up any potential confusion between aspects and facets.

5.3.5 Facet projection

Architecture specification usually leads to construct large and overloaded meta-models due to the high complexity of architectures. This situation pushed us to find out a new perspective to deal with this complexity. Our perspective consists of slicing the architecture (specified by meta-models) into a set of interrelated facets according to stakeholders' quality attributes to streamline architecture evaluation. It is the responsibility of architects to decide which facets to extract to fulfill the requirements of the evaluation task ahead. We call this extraction procedure, *facet projection*. The latter uses practically the model projection algorithm and parameterized projection defined in (Kadri et al., 2020, 2021a) to extract facets automatically or semi-automatically. In this context, model projection allows reducing complexity and downsizing models iteratively to extract elements of interest through projection queries. It should be noted that the framework can embrace other meta-models slicing techniques (Blouin et al., 2015) if they fulfill the architects' needs.

Definition: facet projection

A facet projection is a procedure that transforms a portion of the source meta-models that specify the overall architecture into a new reduced destination meta-model that represents the facet. The transformation consists of extracting from the source model only elements of interest through executing a projection query. This query takes the projection (slicing) criterion as input and produces a facet as output. More specifically, the extracted elements constitute the architectural view. The latter is weaved with the relevant quality view (which depicts the quality attribute inputted as projection criterion) in order to construct the requested facet. A facet projection is a unary operation written as $\prod_{P}(M)$, where:

- P is the projection query.
- M is the source meta-models.

The result of such projection is the facet AF, which means that $\prod_{P}(M) = AF$, and $AF \subseteq M$.

Definition: projection query

A projection query (Kadri et al., 2020) permits the interrogation of source meta-models in order to extract the desired elements. These meta-models are extracted normally from the NFR catalog. In this context, a projection query is a sort of statement that communicates with meta-models to perform projections according to the input parameter. This parameter holds the projection criterion, which is the quality attribute inputted into MS-QuAAF. Inspired by SQL queries, a projection query **p** has the following syntax:

"***project*** *elements* ***from*** *source_meta-model* ***into*** *new_facet* ***where*** *projection_criterion=input parameters".*

This query is partially similar to the SQL selection query and it indicates to the facet projector (figure 5.4) to select elements of interest from the source meta-model and project them to construct a new architecture facet with respect to the projection criterion. The produced facet will be inputted into the quality evaluator module (a set of evaluation services) to proceed with the evaluation process.

We have defined projection queries for the first time in (Kadri et al., 2020, 2021a) to extract sub-models from larger multi-view models. The projection query in this case can hold multiple and different criteria, such as architectural styles, sub-systems, and components. However, since facet projection is a special case of model projection, the projection query uses one criterion, which is the quality attribute under assessment. For example, if we want to extract the facet mapped to the quality attribute *performance*, then the query can be written as follows.

"***project*** *elements* ***from*** *source_meta-models* ***into*** $AF_{performance}$ ***where*** *quality_attribute=performance".*

This query means to extract all architectural decisions taken to promote performance, for instance, introducing concurrency and improving bandwidth in a web-based architecture. The extracted decisions constitute the architectural view, which is weaved to the relevant quality view to construct the performance facet.

More specifically, the facet $AF_{performance}$ is constructed by executing the following four-step algorithm:

Step 1. Reading the source specification of the source meta-models that reside in the NFR catalog.
Step 2. Reading and compiling the projection query p.
Step 3. Extracting the designated elements by the query p to construct the architectural view $AV_{performance}$.
Step 4. Constructing the requested facet by weaving the architectural view $AV_{performance}$ constructed in step3 to the quality view $QV_{performance}$.

Figure 5.4. The facet projector input/output

Slicing tools and model projection have been addressed by many works in the literature. Blouin et .al (Blouin et al., 2015) have used the model slicing technique to build an interactive visualization tool that permits to focalize on the meta-model elements of interest. They used model slicing inspired by the program slicing technique to extract a subset of elements from source models. A DSL called *Kompren* is used to define model slicers to facilitate the creation of such visualization tools. The slicing technique has been used to reduce typical task effort in terms of time, correctness, and navigation.

Androutsopoulos et .al (Androutsopoulos et al., 2011) have used model projection in state-based models to minimize and specialize models for a specific operating environment. They defined four projection algorithms to address three critical concerns, which are correctness, effectiveness, and efficiency. Meyer et al. (Meyer et al., 2006) proposed *Mondarian*, a visualization tool that uses scripts (queries) on the data model to produce specialized views. The framework does not provide interaction facilities. Instead, it represents a meta-model for different visualization tools, in which scripts are used to transform the data model into the visualization model. In (Glinz et al., 2002), the authors proposed *Adora*, a modeling tool that relies on semantics zooms to visualize the desired level of details. In this context, filters are used to reduce the size and complexity of the underlying models.

Most of these works focus on slicing UML models and meta-models, and thus they are dependent to one modeling language, which is UML. On the other hand, the concept of our projection algorithm adopted by facets projection is general and can be applied to any model or meta-model despite their specification language and creation tool by using high-level and customizable projection queries.

5.3.6 An illustrative example of facet projection

To clarify the concept of facets, let us take an example of the server of a Web auction application, which is one of two case studies that will be presented in the last chapter. The server adopts the J2EE technologies to offer services to end-users, which means that its overall architecture is organized as a set of layers. Each layer encompasses a coherent set of related modules, in which messages are sent from the top layer to lower ones.

To specify the architecture of this J2EE server, we have used *Alloy*, which is a lightweight formal modeling language. This language is reinforced by the Alloy analyzer that uses SAT solvers to enable automatic and bounded model checking (Cunha et al., 2015; Jackson, 2012, 2019). The analyzer allows discovering flaws as early as possible during the design time to prevent defects at further development stages. The specification consists of defining a set of rules and constraints through Alloy *predicates* and *facts* to promote the desired quality attributes (listing 5.1). Facts are constraints assumed always to hold, whereas predicates are named constraints with zero or more declarations of arguments.

The Alloy specification shown in listing 5.1 illustrates that an authenticated bidder can search, buy, and pay for goods. These operations are performed by sending messages starting from the top layer down to the lower one. The top layer encompasses java Servlets that allows a bidder to log to his/her account, search, buy, and pay for bids. Each Servlet calls the appropriate EJBs (Enterprise Java Beans are server-side components that encapsulate business logic) packaged within the second layer. An EJB calls in its turn the corresponding java classes that reside in the bottom layer to connect to the auction database to execute users' queries.

```
1 module J2EE_Server2

3 abstract sig Component {}
4 abstract sig layer {contains: some Component}
5 abstract sig Servlet_auction extends Component{calls:Ejb_auction}
. . .
13 //layer03
14 one sig Presentation_Layer extends layer {}
15 sig login extends Servlet_auction {}
16 sig Account {searches:search_article, buys:buy_article, pays:pay_article}
17 sig search_article extends Servlet_auction {}
. . .
21 //Layer02
22 one sig Business_Layer extends layer {}
. . .
26 one sig pay_Ejb extends  Ejb_auction  {pays:pay_DB}
27
28 //Layer01
29 one sig Data_Layer extends layer {}
30 one sig DB extends Service_auction{}
. . .
34 sig pay_DB extends Service_auction{update:article some->Connection_DB}
36 ///The presentation layer must contains only Servlet Components
37 fact {all s:Servlet_auction, e:Ejb_auction, r:Service_auction,
l:Presentation_Layer |s in l.contains && e not in l.contains  && r not in
l.contains}
…
39 ///The Business layer must contains only EJBs Components
40 fact {all s:Servlet_auction, e:Ejb_auction, r:Service_auction,
l:Business_Layer|e in l.contains && s not in l.contains  && r not in l.contains}
41 /*The Data access layer must contains only  Components that provide
          DB connections, login, and execution of users' queries*/
43 fact {all s:Servlet_auction, e:Ejb_auction, r:Service_auction, l:Data_Layer|r
in l.contains && s not in l.contains  && e not in l.contains}…
45 // a bidder can buy only items that can be found in the database
46 fact {all s:search_DB |all b:buy_DB | b.update in s.finds} …
. . .
55 // if a bidder buy some items that's means he found those item in the auction
database
56 fact {all a:Account, s:search_DB| some a.buys implies some s.finds}

//security
fact {all  a: Access_Manager, p:Previlege  |p in a.assign}
fact {all s:Security_Policy, a: login | some a.follow implies some s.identify &&
 #follow.Security_Policy=#s.identify}
. . .
```

Listing 5.1. An excerpt of the server Alloy specification

This specification may consist of hundreds of lines because it defines constraints for a set of quality attributes, such as maintainability, performance, and security. For example, defining the number of layers, the type of components that should reside in each layer, and the relation between those components represent a part of the constraints specified to promote maintainability. The higher the number of quality attributes promoted by the architecture, the more complicated the specification and the larger the size of the resulting meta-models. In this case, we can benefit from the concept of facets to reduce complexity in order to assess the relevant quality attributes. The example shown below is a simplified illustration of the extraction and construction of the *security* facet (mapped to the security attribute).

Example: extracting the security facet
Like all online systems, a web auction can be a target of phishing attacks that acquire user information by masquerading as a legitimate entity. Architects must guarantee the invulnerability of their system against this type of online attack (and other attacks) by taking the appropriate architectural decisions that ensure the achievement of the security attribute. To assess whether this attribute is achieved, the facet projector extracts the architectural parts designated to promote this attribute through the projection algorithm. These parts constitute the security facet. More specifically, the projection is performed by applying the above-mentioned algorithm, as follows.

i. Specifying the projection query destined to interrogate the overall architecture as follows. "**project** elements **from** main_architecture into $AF_{security}$ **where** QA=*Security*".

ii. Extracting the design information mapped to the security attribute, such as the architectural elements, and the security tactics and rules, then saving them in a separate Alloy module (Listing 5.2). This module represents the architectural view of the security facet.

iii. Extracting the quality view of the security facet and saving it in a separate Alloy module.

iv. Constructing the security facet by weaving the above views using a third Alloy module.

The first Alloy module (Listing 5.2) represents the architectural view of the security facet (yellow boxes shown in figure 5.5). It depicts the architectural components needed to secure the login to the system, such as the login, the security policy, the authentication, the authorization, and the access manager components. It also defines a set of security rules that constrain and control users' access to the online auction (lines from 11 to 20). These security tactics constitute the first line of defense against malicious users. A bidder should be authenticated by entering a correct password, a captcha, and/or a digital certificate. If the bidder is properly identified, then the access manager component authorizes him/her to use the auction system by assigning him/her a set of privileges.

```
1 module Security
2 sig login {follow: Security_Policy}
3 one sig Security_Policy {identify: login -> Authentication, adhere:
Password_Policy}
4 one sig Authentication {authorize: login one-> one Authorization}
…
9 enum Protection_Type {password,capcha,degital_certificate}
…
15 //If the security policy identifies an actor, then it should authorize
him/her
16 fact {all s:Security_Policy, a: Authentication | some s.identify  implies
some a.authorize }
17 //if a login is authorized then the system should assign to it a
privilege.
18 fact {all s:Access_Manager, a: Authentication | some a.authorize implies
some s.assign && #a.authorize=# s.assign}
20 pred protect [l:login, p:Password_Policy]{some l implies some l.follow  }
21 run protect {}
```

Listing 5.2. An excerpt of the architectural view of the security facet

The second Alloy module (listing 5.3) represents a simplified quality view of the security facet (grey boxes shown in figure 5.5). This view depicts the security attribute and its interested stakeholders, which are architects, developers, and clients. It illustrates also the role of each stakeholder in the process of satisfying security. In this view, we assume the absence of security sub-attributes.

```
1 module QV_Security2
2 abstract sig  QA {name:String,  To_Satisfy: some Stakeholder}{}
3 sig Stakeholder {name:String, role:lone String}
```

Listing 5.3. The quality view of the security facet

The third Alloy module (listing 5.4) is the one in charge of constructing the security facet by attaching the architectural view to the quality view through the Alloy predicate called *join* (line 4 to line 9). Executing this predicate (line 10) will construct the security facet and give us its graphical representation shown in figure 5.5. The left grey side represents the quality view, whereas the right yellow side represents the architectural view. The graphical representation contributes to making the facet more readable and understandable because it explicitly illustrates the mapping between quality attributes and the architectural decisions taken to promote them.

```
1 open Security as Arch_View
2 open QV_Security2 as Quality_View
3 one sig QA1 extends QA{promotedBy:Security_Policy}
4 pred join[q1:QA1,s1,s2, s3:Stakeholder]{q1.promotedBy  in
Security_Policy
5  q1.name="Security"
6 s1.name="Architect"  s1.role="specify security tactics"
7 s2.name="Client" s2.role="get satisfied"
8 s3.name="developer"   s3.role="implementing security tactics"
9 s1+s2+s3 in q1.To_Satisfy}
10 run join for   3
```

Listing 5.4. Weaving the architectural view to the quality view

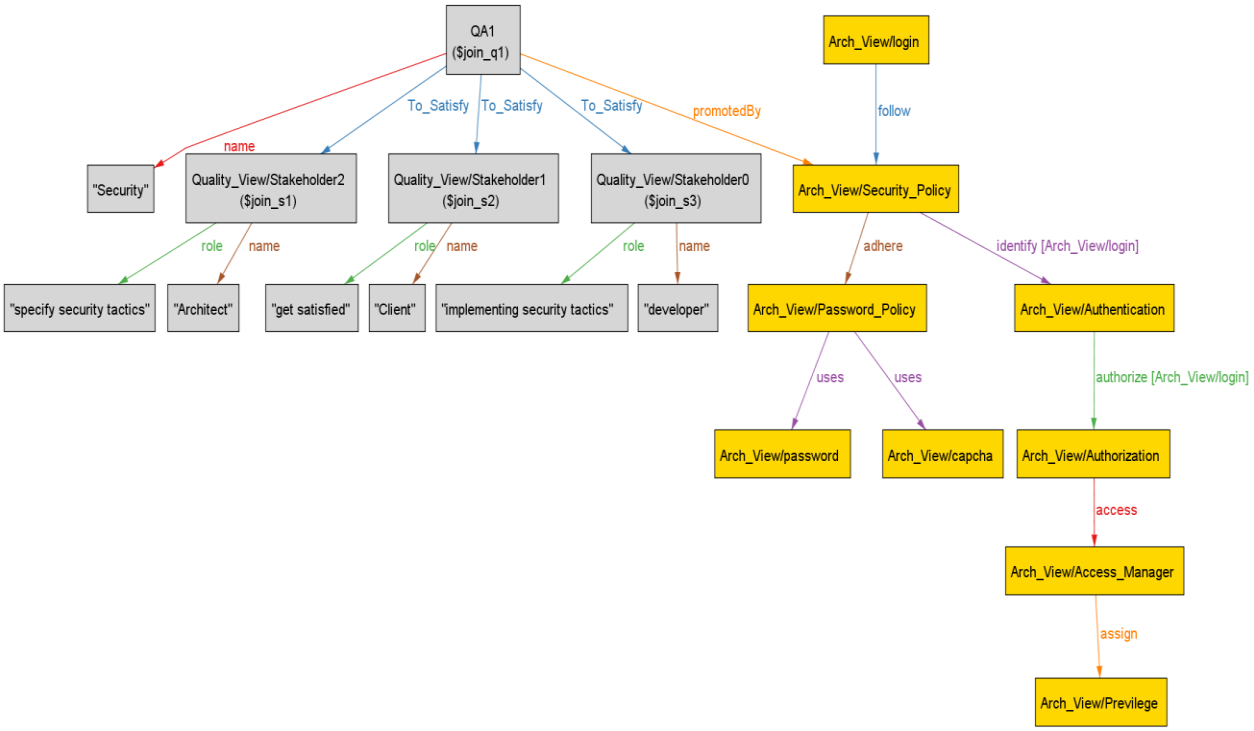

Figure 5.5. The graphical representation of the security facet

5.3.7 Expected properties of the generated facets

Although we have used Alloy to illustrate the construction of a facet, the concept of projection is independent of any specification language and can be applied to architectures specified with other languages. However, despite the specification language, the produced facets must adhere to the following properties:

- *Navigability*: This property applies to facets specified with languages that provide graphical representation support. A produced facet must provide easy navigation due to its reduced size.
- *Clarity*: A facet must express explicitly and clearly the mapping between the quality attribute of interest and the relevant architectural decisions.
- *Correctness*: A produced facet must adhere to the projection query that specifies the architectural information supposed to be extracted.

The model projection paradigm has proved its effectiveness in reducing the navigation time and reducing the size and complexity of the produced models (Kadri et al., 2021a), which can streamline significantly the next task of the evaluation process. However, it is possible to encounter some correctness issues when constructing a facet. This can probably happen due to ambiguous architecture specification. Therefore, it is preferable to revise and double-check the correctness of the facet before sending it to the quality evaluator to obtain reliable and more accurate assessment results.

5.4 Conclusion

In this chapter, we have presented the first module of MS-QuAAF, which is the facet projector. The first phase of quality assessment using MS-QuAAF is performed through this module, in which architecture facets are extracted from the main architecture specified by large meta-models. An architecture facet is proposed as a means to hide unnecessary architecture information and show only the elements of interest related to the quality attribute inputted into the framework. Facets are extracted through facet projection, which is an instance of model projection. The latter was proposed for the first time in (Kadri et al., 2020) to extract elements of interest from oversized source models using projection queries. In this separate work; projection queries can accept multiple types of projection criteria, and therefore extracting the desired elements according to these criteria. Subsequently, we have proposed in another work (Kadri et al., 2021a) an improvement of the previous algorithm using incremental projection, in which extracting elements can be performed iteratively through many steps.

Facet projection is a special case of model projection, in which the projection query holds one criterion, which is the quality attribute under assessment. Accordingly, only architectural decisions related to this attribute are extracted, which allows reducing complexity and

navigation time of large meta-models, and thus boosting up understandability and facilitating quality assessment.

Once facets are extracted, the second phase of quality assessment commences through the quality evaluator module. The latter will be explained thoroughly in the next chapter.

Chapter 6

The MS-QuAAF's evaluation services and metrics

In the previous chapter, we have presented an introduction to the evaluation methodology followed by the framework MS-QuAAF and the rationale and motivation behind such methodology. The framework consists of two primary modules, which are the facet projector and the quality evaluator. The former has been already depicted in detail in the preceding chapter. The latter will be presented in the current chapter.

In this chapter, we will present the evaluation effort performed through the quality evaluator module. The latter consists of three evaluation services each of which defines a set of generic metrics calculated mathematically through sequences of equations. Each service provides an evaluation sub-report as output, which contains the assessment results of the corresponding development stage (design or implementation). Finally, a global report is generated at the end of the evaluation process.

Contents

6.1 The evaluation effort 115

6.2 The quality evaluator module 117

6.3 Conclusion 132

6.1 The evaluation effort

Once the desired facets are extracted from the overall architecture, they will be inputted into the quality evaluator module to proceed with the quality evaluation. The process starting from projecting facets to generating the final quality report is called the evaluation effort. The generic AE (Architecture Evaluation) framework defines the evaluation effort by the activity that determines the actual value of architecture. The evaluation effort can be performed at many stages during the development entity (chapter 5). Accordingly, MS-QuAAF allows performing this effort through a set of services at two main development stages, which are the conceptual design and the implementation stages. In the context of MS-QuAAF, the AE effort consists of

a) Accepting any quality attribute as input.
b) Extracting the corresponding architecture facet.
c) Calling the evaluation services to perform the assigned assessment tasks using the appropriate metrics.

d) Generating the evaluation report as output.

However, it should be noted that the framework is designed to assess only one quality attribute at a time on the condition that each attribute has an architectural specification in the NFR catalog. The defined metrics are common to all attributes, which means that despite the quality attribute inputted into the framework, the quantitative measurement is performed using the same set of metrics through the following services (Figure 6.1):

1) The first service called RDA (Rules Defectiveness Analysis) is dedicated to assessing the defectiveness of the design rules encompassed within the inputted facet.
2) The second service called RTA (Responsibilities Tree Assessment) is dedicated to assessing the implementation and the achievement of the NFR responsibilities assigned to promote the attribute being assessed.
3) The third service called LAFA (Late-assessment and final analysis) is dedicated to finalizing the evaluation effort and generating the final assessment and analysis report.

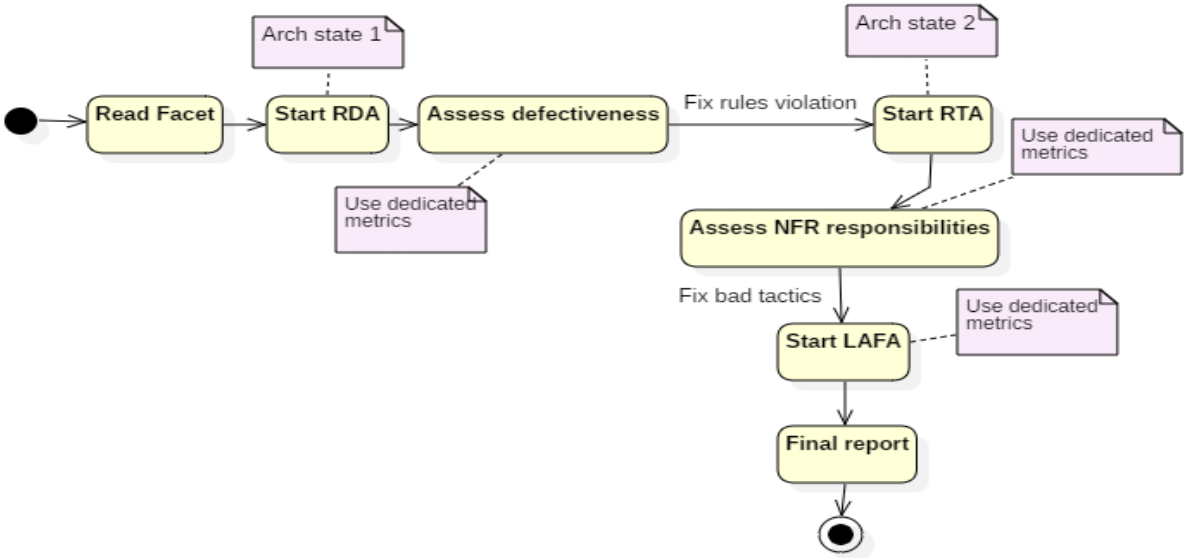

Figure 6.1. MS-QuAAF evaluation effort

Since the evaluation effort is performed at two development stages, two well-defined architecture states are assessed by the framework. These states can be defined as follows.

(i) *State 1*: the first state appears after the design of the architecture has been completely accomplished. This state is called also the designed architecture.

(ii) *State 2*: the second state emerges after the system has been implemented. This state is called also the implemented architecture.

Services	Architecture state	Evaluation type
RDA	State 1	Pre-assessment
RTA	State 2	Post-assessment
LAFA	State 1, State 2	Conclusive post-assessment

Table 6.1. MS-QuAAF's Services

These states are assessed by the evaluation services shown in table 6.1. The first service is designed for pre-evaluating the architecture when it emerges in the first state. The second service is capable of post-evaluating the architecture when it occurs in *state 2*. The third service uses the results returned by the previous services to output the final evaluation report. The evaluation of these states aims to provide stakeholders with clear and quantitative results to keep them updated on the current state of the architecture quality, and thus performing the necessary elaboration to boost up this quality.

The defined architecture states occur after the accomplishment of the design and implementation stages, respectively. According to (Abowd et al., 1997), the succession of these stages constitutes the phase called *post-deployment*, which is the last phase of interest to architecture evaluation. In our turn, we have divided this phase into two steps. The first step is called *pre-assessment,* and it is carried out by the service RDA. The second step is called *post-assessment,* and it is performed by the services RTA and LAFA. The framework is not concerned with the evaluation phases called *early* and *middle* because the architecture is still immature and needs some iterative elaboration. However, in the post-deployment phase, the architecture is completely mature, and it is ready to answer the following questions:

a) Are the stakeholders' non-functional requirements met?

b) Are the architectural decisions taken effective?

c) Does the designed architecture match the specified architecture?

d) Does the implemented architecture match the designed architecture?

6.2 The quality evaluator module

In this section, we will present the second module of the framework, which is the quality evaluator module (figure 5.1). The latter consists of three evaluation services, each of which provides a set of evaluation metrics. The succession of these services constitutes the architecture evaluation process that starts after the facet projection process.

6.2.1 Rules Defectiveness Analysis service (RDA)

Design rules are the basic elements and the cornerstone on which we build architectural specifications. They represent instructions and constraints at the meta-model level (Kallel et al., 2018) to indicate how architecture elements should be organized, related, and behaved to meet functional and non-functional requirements. From this perspective, NFR rules represent a subset of design rules dedicated only to promoting stakeholders' quality attributes. This type of rule is the only type analyzed by the framework's services. In the rest of this chapter, we use the term rules instead of NFR rules for the sake of simplicity and brevity.

RDA is an analyzing service called to gauge the defectiveness of rules specified at the early stages. A rule defectiveness measure is a set of metrics used to calculate the degree or the density of rules violation of software architecture when it occurs in state 1. Rules violations are

design infringements committed by designers when they deviate from the instructions specified in the NFR catalog. The reason behind introducing this service is the need for applying an earlier analysis that assesses the designed architecture before going any further through the development process. This would help designers to:

- Fixing irregularities and non-conformance to design standards and architectural decisions in order to reduce architecture defectiveness. This includes architectural styles, patterns, and any type of rule specified to promote quality attributes.
- Eliminating or decreasing undesired architecture complexity.
- Reducing reworks on architecture during and after the implementation stage
- Predicting the quality of the next state of architecture before proceeding with the implementation stage.

RDA adopts a weight-based approach to assess rules defectiveness. It uses an importance scale to cluster rules into a set of macro-groups. Each macro-group is assessed separately, and then an evaluation recap that contains the defectiveness percentage, the number of rules infringements, and the violated rules will be produced at the end of the analysis.

6.2.1.1 Macro-groups

A macro-group is a cluster of rules that have the same weight and promote the same quality attribute. The grouping is based on a customizable scale that uses the importance degree (extreme, strong, moderate…etc.) and the violation severity (catastrophic, major, minor…etc.) of each involved rule. Grouping rules into macro-groups reduces significantly analysis iterations and the time spent to calculate architecture defectiveness. More specifically, instead of calculating facet defectiveness using the fulfillment score of each rule, we use only the score of each macro-group (eq. (2)). If we have, for instance, fifty rules grouped into four macro-groups then, the number of iterations to calculate the defectiveness is reduced to four instead of fifty (eq. 5). More formally, we represent a macro-group MGi by the couplet <Ri, Wi>, where:

- Ri= {rij /j =1, 2...n} is the set of n rules constituting the macro-group MGi.
- Wi is the weight of the macro-group MGi, where $0 \leq Wi \leq 1$.

6.2.1.2 Calculating architecture defectiveness metrics

Design-time architectural defects represent the parts of the design that does not comply with architectural decisions taken at earlier stages and specified at the meta-model level. They represent the set of all rules violated when the design of architecture has been accomplished (state 1). An architectural defect has the following properties:

- Source: a defect is caused by the violation of a design rule.
- Severity: the degree of damage that a defect can cause. It has a direct correlation with the weight of rules. The larger the weight of rules is, the higher the severity of their defectiveness.

RDA calculates the fulfillment score of each macro-group separately then, it calculates the defectiveness metrics of each facet inputted into MS-QuAAF, and finally, it calculates the overall architecture defectiveness using the following equations:

Definition 1: Macro-group fulfillment
Let $MG_i = \{R_1, R_2,...,R_n\}$ be the set of rules of the macro-group MG_i, W_i be the weight of MG_i, and $S=\{S_1, S_2,..., S_n\}$ be the set of fulfillment scores in which, each score is assigned to each rule of MG_i, respectively, as follows.

$$Si = \begin{cases} 1, & Ri\ is\ not\ violated \\ 0, & Ri\ is\ violated \end{cases} \quad (1)$$

This procedure requires architects to perform design reviews to identify the violated rules of each macro-group MG_i then; RDA calculates the fulfillment score of each MG_i as follows.

$$MGF_i = \left(\sum_{i=1}^{n} S_i\right)\frac{W_i}{n}, \quad (2)$$

where MGF_i represents the satisfaction score of the macro-group MG_i. Introducing the weight W_i allows obtaining more accurate scores. In this context, violating rules with a larger weight can increase significantly the architecture defectiveness compared to rules with a smaller weight.

Calculating MGF_i of all macro-groups of the facet AF_i makes determining the satisfaction score of the related quality attribute QA_i much easier (eq. (3) and eq. (5)) It suffices to sum the obtained MGF_i to calculate this score. Subsequently, the overall fulfillment score is obtained easily by summing the scores of all stakeholders' quality attributes (eq. (4) and eq. (6)).

Definition 2: the **QuARF** (Quality Attribute Rules Fulfillment) metric
Let $MG= \{MG_1, MG_2,\ldots, MGn\}$ be the set of macro-groups of the facet AF_i, MGF_i be the fulfillment score of each MG_i of MG, and QA_i is the quality attribute promoted by AF_i, then

$$QuARF_{QAi} = \sum_{i=1}^{n} MGF_i \quad (3)$$

is the metric used to measure the fulfillment of rules of the facet AF_i. More accurately, each quality attribute QA_i is pre-assessed by measuring the conformance of the corresponding facet AF_i to the architectural decisions taken. This metric is calculated by summing the score of each macro-group (calculated by eq. (2)) of MG. It is obvious that $0 \leq QuARF_{QAi} \leq 1$, which means that the closer the value of $QuARF_{QAi}$ is to 1, the smaller the number of rules violations.

Example: let AF_1 be the architecture facet mapped to the *performance* attribute. Let us assume that the total number of rules specified within AF_1 is 25 rules scattered over 3 macro-groups:

MG_1 contains 8 strongly important rules, MG_2 contains 12 moderately important rules, and MG_3 contains 5 slightly important rules. Assuming the weight of each macro-group is 0.5, 0.3, and 0.2, respectively (can be calculated by eq. (7), eq. (8), and eq. (9)), and the number of violated rules is 2 rules in MG1, 4 rules in MG2, and 2 rules in MG3, then the satisfaction score of each macro- group is calculated using eq. (2) as follows.

$MG_1 = (1+1+1+1+1+1+0+0)*0.5/8=0.375.$
$MG_2 = (1+1+1+1+1+1+1+1+0+0+0+0)*0.3/12=0.2.$
$MG_3 = (1+1+1+0+0)*0.2/5=0.12.$ The obtained scores are used to calculate the QuARF of performance as follows.

$QuARF_{performance} = 0.375+0.2+0.12 \approx 0.7$. The weight of macro-groups has a big influence on the satisfaction score of each quality attribute. Satisfying strongly important rules can enhance significantly these scores. For example, if all rules of MG1 are satisfied, then $QuARF_{performance}$ is increased to 0.82, whereas the optimal value is 1.

Definition 3: the **OVF** (Overall Fulfillment) metric
Let $QA= \{QA_1, QA_2,\ldots, QA_n\}$ be the set of all stakeholders' quality attributes, $AF= \{AF_1, AF_2,\ldots, AF_n\}$ be the set of the corresponding facets of the architecture SA, and $QuARF_{QAi}$ be the fulfillment score of each AF_i, then

$$OVF = \frac{\sum_{i=1}^{n} QuARF_{QAi}}{n} \quad (4)$$

is the metric used to measure the overall rules' fulfillment of SA. OVF is calculated by summing the score of each facet AF_i of AF (calculated by eq. 3). Obviously, $0 \leq OVF \leq 1$, which means that the closer the value of OVF is to 1, the smaller the architecture defectiveness, and vice versa.

Definition 4: the **QuARD** (Quality Attribute Rules Defectiveness) metric
Let $QuARF_{QAi}$ be the fulfillment score of the facet AF_i, then

$$QuARD_{QAi}\ \% = \left(1 - QuARF_{QAi}\right).100 \quad (5)$$

is the metric used to measure the defectiveness percentage of the quality attribute QA_i promoted by the facet AF_i. The smaller the value of $QuARF_{QAi}$, the larger the defectiveness of the facet, which affects negatively the quality attribute promoted. In the example above, the QuARD of performance is equal as follows.

$QuARD_{performance} = (1\text{-}0.7)*100 = 30\%.$

Definition 5: the **OVD** (Overall Defectiveness) metric
Let OVF be the overall fulfillment score of the architecture SA, then

$$OVD\ \% = (1 - OVF).100 \quad (6)$$

is the metric used to measure the overall defectiveness percentage of SA. The smaller the value of OVD, the healthier the state of the designed architecture.

6.2.1.3 Macro-groups weightage

Architectural rules specified by architects play explicit and implicit roles in achieving specific quality attributes. The weight of a rule is correlated with the importance of its role in the stakeholders' viewpoint. For Example, favoring modifiability over performance means that rules promoting modifiability weigh more than performance's rules. Consequently, violating rules that determine how layers should communicate with each other in a layered design is not the same as violating rules that recommend the maximum number of defined layers. In the former, the violation leads to destroying a design principle that keeps a layered style consistent, which affects modifiability directly. In the latter, the violation may affect performance, which has less priority than modifiability. The violation severity permits determining the weight of rules specified to promote a quality attribute. The larger the weight of rules, the higher the violation severity and the defectiveness ratio.

Attributing weight to rules is not a trivial task because it depends on many criteria, such as quality attributes prioritization, trade-offs, and violation depth. Hence, this allows us to treat the calculating of macro-groups' weight as a decision-making problem. As we have mentioned above, rules are categorized into macro-groups according to architects' judgment that uses a qualitative importance scale. However, this categorization does not assign quantitative weights to macro-groups. For this reason, RDA uses the AHP (Analytic hierarchy process) (R. W. Saaty, 1987; T. L. Saaty & Vargas, 2012; Svahnberg et al., 2003) technique to calculate the weight of each defined macro-group.

AHP is a multi-criteria decision-making technique that assigns preference values to alternatives. Such values are used to help decision-makers in ranking and selecting the best choice that suits their goals by decomposing hierarchically the decision problem into easier sub-problems (Svahnberg et al., 2003). The hierarchy elements are evaluated systematically using a pairwise comparison matrix along with expert judgments. The matrix is created with the help of a quantitative scale of relative importance called the Saaty scale (Table 6.2).

Values	Importance levels
1	Equal importance
3	Moderate importance
5	Strong importance
7	Very strong importance
9	Extreme importance
2, 4, 6, 8	Intermediate values
1/2,…, 1/9	Inverse comparison

Table 6.2. The Saaty scale

The matrix A $_{(nxn)}$ contains the pairwise comparison values between macro-groups calculated using the Saaty scale's importance levels, where A $_{(nxn)}$ is written as follows.

$$A = \begin{array}{c} \\ MG_1 \\ \vdots \\ MG_n \end{array} \begin{array}{c} \begin{array}{ccc} MG_1 & \cdots & MG_n \end{array} \\ \begin{pmatrix} A_{11} & \cdots & A_{1n} \\ \vdots & \ddots & \vdots \\ A_{n1} & \cdots & A_{nn} \end{pmatrix} \end{array} \tag{7}$$

Each value of A is normalized by the sum of each column as follows.

$$A_{ij} = \frac{A_{ij}}{\sum_{i=1}^{n} A_{ij}} \tag{8}$$

After the matrix has completely normalized, we calculate the weight of each macro-group by dividing the sum of each line by *n* as follows.

$$W_i = \frac{\sum_{j=1}^{n} A_{ij}}{n} \tag{9}$$

The resulting weights are used to calculate the fulfillment score of macro-groups depicted by eq. (2). The weightage of macro-groups permits deducing the violation severity of each involved rule, and thus helping RDA in calculating architecture defectiveness more equitably (by giving each rule the weight that deserves) and accurately.

In summary, the RDA service performs the defectiveness analysis and assessment as follows.

Step 1. Dividing the inputted facet into a set of macro-groups using a qualitative importance scale.
Step 2. Calculating the weight of each macro-group quantitatively using the AHP method (eq. 7, 8, 9).
Step 3. Calculating the fulfillment of macro-groups, facets, and the overall architecture (eq. 1, 2, 3, 4).
Step 4. Calculating the defectiveness of macro-groups, facets, and the overall architecture (eq. 5, 6).
Step 5. Generating the defectiveness sub-report.

6.2.2 Responsibilities Tree Assessment service (RTA)

Although RDA service permits us to analyze rules infringements and calculates architecture defectiveness at the end of the design stage (state 1), it does not allow us to assess the satisfaction of these rules after the accomplishment of the implementation stage (architecture state 2). For this purpose, a second service called RTA is introduced into MS-QuAAF to provide a continuous architecture evaluation. RTA defines a new type of metric called RSI (Responsibilities Satisfaction Indicator) to assess the satisfaction of NFR responsibilities

assigned to promote the inputted quality attribute by performing a responsibility decomposition analysis.

Definition 6: the RSI metric
RSI is the fundamental assessment metric defined within MS-QuAAF. It is an indicator used to gauge the satisfaction of NFR responsibilities by applying a responsibility decomposition analysis.

A responsibility decomposition analysis consists of creating, analyzing, and assessing quantitatively a responsibilities tree called the Responsibilities Satisfaction Tree (RST). The latter is created by decomposing each node (responsibility) into several sub-responsibilities recursively until no further decomposition is possible. The service RTA uses RST to calculate the satisfaction of the top main responsibility (e.g., enhancing performance) by calculating the satisfaction of each weighted sub-responsibility starting from the bottommost nodes of the tree (bottom-up assessment).

6.2.2.1 The Responsibilities Satisfaction Tree (RST)

The responsibility decomposition analysis is inspired by the *goal decomposition analysis* used by the NFR framework originated by (Chung et al., 2000). The NFR framework is centered upon the concept of *Softgoal*, which is the basic unit for representing non-functional requirements (see chapter 4). A Softgoal is a goal that does not have a clear-cut criterion for its satisfaction (Mylopoulos et al., 1999), such as security, safety, and reliability. Currently, there are three main types of goal decomposition analysis (Kobayashi et al., 2016; Yamamoto, 2015; Zhou et al., 2020): the Softgoal Interdependency Graph (SIG) (Chung et al., 2000), the Fault Tree Analysis (FTA) (Ruijters & Stoelinga, 2015), and the Goal Structuring Notation (GSN) (Kelly & Weaver, 2004). However, these methods use a qualitative evaluation of Softgoals, and they cannot evaluate the satisfaction of these goals quantitatively. Besides, these types of goal-driven methods are used to evaluate a specific type of NFR, such as security and robustness, and cannot be used for a generic architecture evaluation. In this work, we propose the service RTA that uses RST to evaluate the satisfaction of the assigned NFR responsibilities quantitatively regardless of the type of quality attributes inputted into the MS-QuAAF framework.

RST (Figure 6.2) is a quantitative responsibility tree inspired by the weighted SIG and goal decomposition tree depicted in (Kobayashi et al., 2016; Subramanian & Zalewski, 2014; Yamamoto, 2015). It uses the same type of SIG's relationships, which are the decomposition and the contribution relationships. However, it uses different types of nodes, which are the top responsibility (root), sub-responsibilities, and tactics (leaves) instead of the top claim, sub-claims, and evidence (or operationalization Softgoals) nodes, respectively. Additionally, the weight of responsibilities is calculated using dedicated equations (eq.10 to eq.15) unlike

quantitative SIGs where weights are assigned to softgoals according to designers' subjective estimation and without providing straightforward methods for calculating those weights.

Formally, an RST is represented by the quintuplet $<R_0, R, T, D, C>$, where:

- R_0 is the top main responsibility, which represents the RST's root.
- $R= \{R_i / i=1...n\}$ is the set of sub-responsibilities that represent children and grand-children of R_0.
- $T= \{T_i / i=1...n\}$ is the set of tactics linked to the bottommost sub-responsibilities nodes. These tactics represent the RST's leaves
- $D= \{Di / i=1...n\}$ is the set of decomposition links that relate responsibilities to each other.
- $C= \{Ci / i=1...n\}$ is the set of contribution links that relate tactics to responsibilities.

The construction of the RST is performed by applying the following steps sequentially.

Step 1. *Top responsibility definition*

The first step to constructing RST is to define the main responsibility assigned to promote a quality attribute. Each attribute is associated exactly with one top responsibility. For instance, RST's main responsibility assigned to promote *performance* can be expressed as *boosting performance.*

Step 2. *Responsibilities decomposition*

Each responsibility is decomposed and refined recursively into sub-responsibilities starting from the top responsibility until the decomposition criterion is fulfilled. This criterion is satisfied when all the produced responsibilities are atomics (indivisible), and no further decomposition can be applied. For instance, the responsibility called *boosting performance* can be decomposed into *adopting multi-tier architecture, managing network latency, avoiding bottlenecks,* and so forth.

Step 3. *Tactics attribution*

After the decomposition has been completely finished, tactics are attributed to the bottommost responsibilities. More specifically, at least one tactic is linked to each atomic responsibility. In this context, a tactic is a set of rules specified to achieve an NFR responsibility. Tactics can contribute positively or negatively to a responsibility achievement, which means that the more the number of tactics is fulfilled, the more responsibilities are satisfied and vice versa. For this reason, links between tactics and responsibilities are called contribution links.

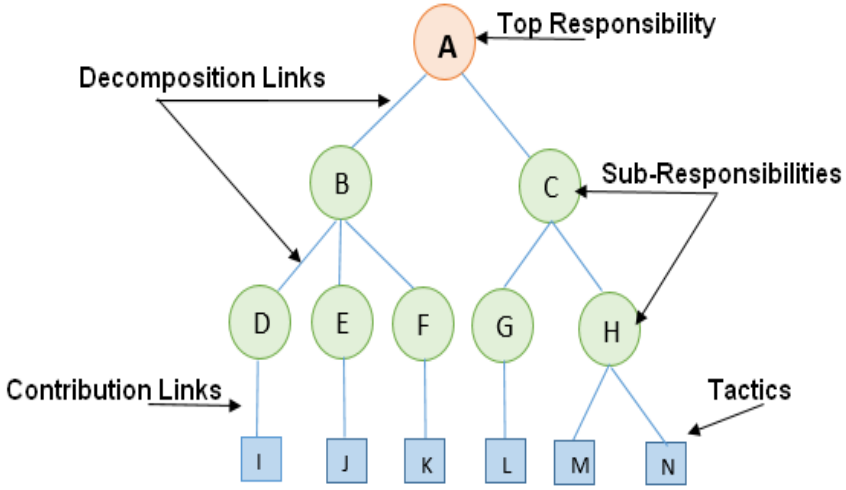

Figure 6.2. The Responsibilities Satisfaction Tree (RST)

6.2.2.2 Quantitative evaluation using RST

Evaluating NFR responsibilities using RST is a multi-step bottom-up process that evaluates the achievement of NFR responsibilities incrementally. It starts by calculating the attribute values of tactic nodes using the weight and the satisfaction score of these architectural tactics. Then, it propagates these values to the upper responsibilities nodes level after level using the contribution and decomposition weights until it reaches the top responsibility (Figure 6.3). More specifically, the evaluation is conducted by applying the following steps.

Step1. *Calculating attribute values for tactic nodes*

Step 1.1. *Assigning satisfaction scores to tactics*

The first step taken to evaluate NFR responsibilities consists of assigning to each tactic node a satisfaction score. The latter one is intended to assess whether the tactics specified at the design stage are well-implemented and satisfied at the implementation stage. The assessment consists of attributing score values to tactics according to an adjustable satisfaction scale (Figure 6.3 (a)). In this work, we use the following values:

- 2: Strongly satisfied.
- 1: Moderately satisfied.
- -1: Unsatisfied.
- -2: Strongly unsatisfied.

Step 1.2. *Calculating the weight of tactics*

A tactic is a mixture of rules that constitutes a design technique. The weight of a tactic is calculated by summing the weights of all its rules. The weight of these rules has been already calculated by the first service RDA when the weight of macro-groups was determined using AHP (section 6.2.1.3). Therefore, the weight of a tactic node is calculated as follows.

Let T_i be a tactic node, R= {R_1, R_2, …R_n} be the set of rules constituting T_i, and RW={RW_1, RW_2,…RW_n} be the set of weights of R, respectively, then

$$TW_i = \sum_{i=1}^{n} RW_i \quad (10)$$

is the weight of the tactic node T_i. For instance, if the tactic T_1 encompasses 5 rules, each of which have the weights {0.5, 0.3, 0.2, 0.2, 0.3}, respectively, then TW_1= 0.5+0.3+0.2+0.2+0.3= 1.5.

Step 1.3. *Calculating attributes values*

Let T= {T_1, T_2,…T_n} be a set of tactic nodes, TW={TW_1, TW_2,…TW_n} be a set of the corresponding tactic weights of T, and S= {S_1, S_2,…S_n} be a set of satisfaction score of T, then

$$TV_i = TW_i * S_i \quad (11)$$

is the attribute value of the node T_i that has the weight TW_i and the score S_i (Figure 6.3 (a)). For example, the tactic node *K* shown in this figure is calculated by multiplying the weight 0.45 by the satisfaction score 2, which gives us 0.45 as the attribute value of *K*.

Unlike quantitative Softgoal-based approaches that depend only on the satisfaction scale to assign values to evidence, we have used the weight of tactic nodes beside satisfaction score to calculate attribute values of these nodes. We believe that using tactic weights in calculating attributes gives us more accurate results than other approaches. More specifically, the weight of tactics has a direct correlation with the positive and negative contribution of these tactics in achieving NFR responsibilities. The larger the weights are, the higher the contribution is, and vice versa.

Step 2. *Calculating attribute values for responsibilities nodes*

Calculating attribute values for responsibilities consists of using the propagation technique. Attribute values can be propagated from tactics nodes to sub-responsibilities nodes or from sub-responsibilities to upper sub-responsibilities.

Step 2.1. *Tactic to sub-responsibility propagation using contribution weight*

Attribute values of tactic nodes can determine how much a tactic can contribute to the achievement of the upper nodes. In the RST tree, an atomic sub-responsibility can be fulfilled by a set of tactics, each of which has a contribution weight. These weights are calculated as follows.

Let RS_i be an atomic responsibility, T= {T_1, T_2,…T_n} be the set of tactics assigned to RS_i, TW={TW_1, TW_2,…TW_n} be the set of tactics' weight of T, and TV={TV_1, TV_2,…TV_n} be the attribute values of T, respectively, then

$$CW_i = \frac{|TW_i|}{\sum_{i=1}^{n}|TW_i|} \quad (12)$$

is the contribution weight of the contribution link relating the tactic T_i with the sub-responsibility RS_i, where $CW_1+CW_2+\ldots+CW_n= 1$.

By using attribute values and contribution weight of tactics, the attribute values of atomic sub-responsibilities can be propagated from these tactics using the following equation.

$$ARV_i = \sum_{i=1}^{n} TV_i * CW_i \quad (13),$$

where ARV_i is the attribute value of RS_i propagated from T by multiplying TV's values by CW's contribution weights (Figure 6.3 (b)). For instance, the atomic responsibility *H* shown in this figure has two contribution weights (0.37, 0.63) linked to two tactic nodes (M, N) that have the attribute values (-0.29, 0.5), respectively. Therefore, the attribute value of H is calculated by propagating the attribute values of *M* and *N* through the contribution weights as follows.

$ARV_H= (0.37*\text{-}0.29) + (0.63*0.5) \approx 0.21.$

Step 2.2. *Sub-responsibility to sub-responsibility recursive propagation using decomposition weight*

As we have mentioned above, a top responsibility node is divided into a set of sub-responsibilities recursively, each of which has a weight called the *decomposition weight*. This weight reflects the importance of a sub-responsibility in achieving its parent node. Accordingly, calculating a decomposition weight is performed as follows.

Let $RS_i= \{SRS_1, SRS_2,\ldots SRS_n\}$ be the set of child sub-responsibilities of the parent responsibility RS_i, and $RV_i= \{SRV_1, SRV_2,\ldots SRV_n\}$ be the set of attribute values of these sub-responsibilities, respectively, then

$$DW_i = \frac{|SRV_i|}{\sum_{i=1}^{n}|SRV_i|} \quad (14)$$

is the decomposition weight of the decomposition link relating RS_i to its child sub-responsibilities, where $DW_1+DW_2+\ldots+DW_n= 1$.

To propagate attribute values from lower sub-responsibilities nodes to parent nodes recursively, we use the following equation:

$$RV_i = \sum_{i=1}^{n} SRV_i * DW_i \quad (15),$$

where RV_i is the attribute value of the parent RS_i propagated from sub-responsibilities by multiplying attribute values of child nodes by the corresponding decomposition weights (Figure 6.3 (c)). For instance, the parent node *C* shown in this figure has two children sub-responsibilities (G, H) that have the attribute values (0.7, 0.21), in which the decomposition weights are 0.77 and 0.23, respectively. Therefore, the attribute value of *C* is calculated by propagating the attribute values of *G* and *H* through their decomposition weights as follows.

$RV_C = (0.7*0.77) + (0.21*0.23) \approx 0.59.$

Step 2.3. *Sub-responsibility to main responsibility propagation, calculating RSI*

In this last step, RTA calculates the metric RSI by propagating attribute values from sub-responsibilities to the top responsibility using the same equations defined in the previous step as follows.

Let $TR = \{SRS_1, SRS_2, \ldots SRS_n\}$ be the set of sub-responsibilities of the top responsibility TR, and $RV = \{SRV_1, SRV_2, \ldots SRV_n\}$ be the set of attribute values of the child sub-responsibilities, and $DW = \{DW_1, DW_2, \ldots, DW_n\}$ be the decomposition weight of RV, respectively, then

$$RSI = \sum_{1}^{n} SRV_i * DW_i \qquad (16),$$

where RSI is the attribute value and the satisfaction indicator of the top responsibility TR propagated from the lower sub-responsibilities (Figure 6.3 (d)).

RSI represents the final evaluation score that assesses the achievement of the NFR responsibilities assigned to promote the quality attribute under assessment. The service RTA compares this score with the score of the best-case scenario (e.g., all tactics are satisfied) and the score of the worst-case scenario (e.g., all tactics are unsatisfied) to indicate the position of the current evaluation in the satisfaction scale.

In the example shown in figure 6.3, RSI is equal to 1.17 in case all tactic nodes are strongly satisfied (optimal) or it is equal to 0.718 when all tactics are satisfied. Consequently, the obtained score of 0.71 is considered satisfied because it is very close to the value of 0.718.

The outcome of assessing NFR-responsibilities is to detect bad-implemented and non-fulfilled tactics, which helps architects in sanitizing the architecture before deploying the system. The sanitization consists of refining the architecture with minor and medium changes in order to fix the emerged defects since carrying out major changes to the architecture at this development stage can be hard to achieve.

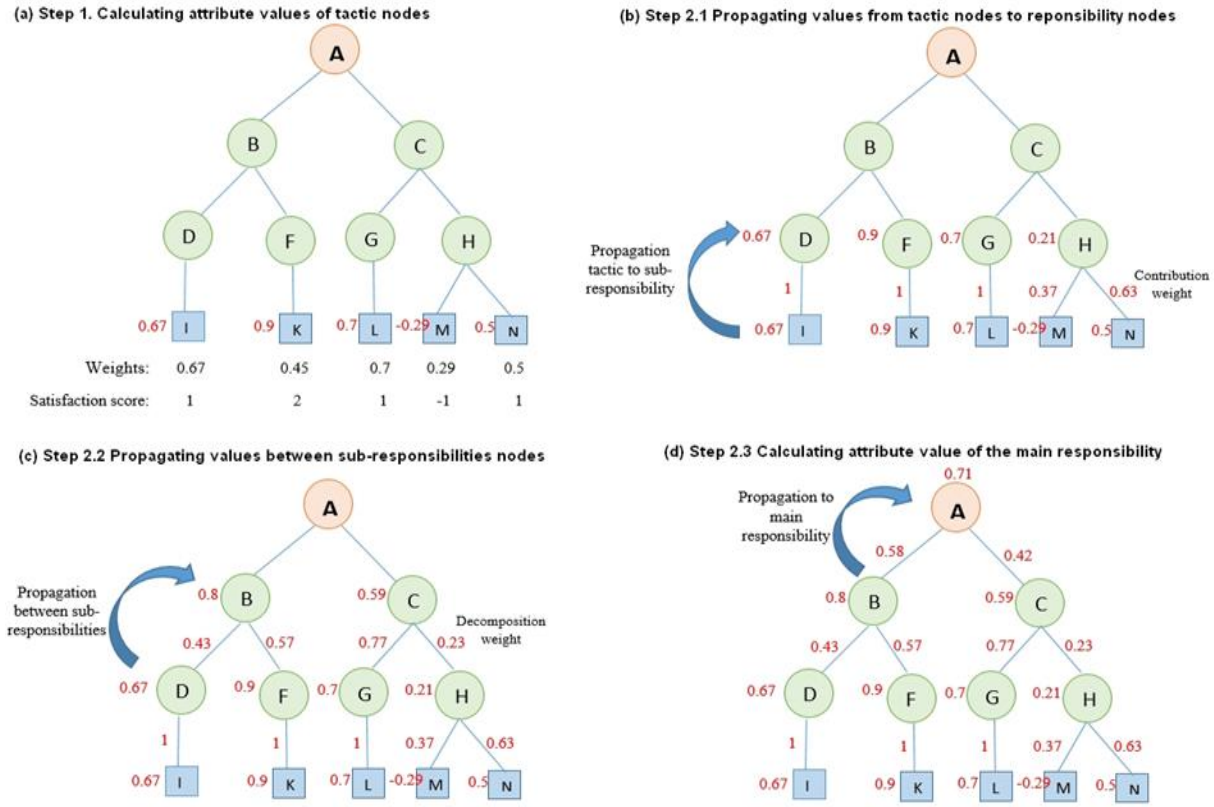

Figure 6.3. An example of a quantitative evaluation using RST

6.2.3 Late-assessment and final analysis service (LAFA)

LAFA is the last service called by MS-QuAAF to finalize the evaluation effort. It consists of generating the final assessment report using dedicated metrics. These metrics are calculated using an evaluation matrix that encapsulates both RDA and RTA's metrics, which means that architecture state 1 and state 2 are covered by the same matrix. The evaluation consists of measuring the closeness between the matrix values that represent the previously obtained assessment results and the optimum values contained in separate vectors called the optimal vectors. The closeness metrics consist of calculating the Euclidean distance (ED) between the matrix and the optimal vectors. The obtained results permit us to make final judgments on architecture quality at both the design and implementation stages and thus finalizing the assessment effort by generating a concluding evaluation report.

6.2.3.1 Evaluation model

The evaluation model consists of constructing a global evaluation matrix that encompasses both architecture states (state 1, state 2), in which alternatives (rows) represent quality requirements, and attributes (columns) represent metrics. More specifically, two metrics are used in this evaluation model, which are QuARF and RSI calculated previously by RDA and RTA. Consequently, the matrix is constructed as follows.

Let

- $QA = \{QA_i / i \in N\}$ be the set of stakeholders' quality attributes, where $N = \{1, 2, \ldots n\}$.

- S= {S_k/ k ∈ P} be the set of architecture states, where P= {1, 2}.
- M= {QuARF, RSI} be the set of metrics used in the final assessment, then

$$X = (Xij)_{nx2} = \begin{matrix} \\ QA_1 \\ \vdots \\ QA_n \end{matrix} \begin{matrix} QuARF & RSI \\ \begin{pmatrix} X_{11} & X_{12} \\ \vdots & \vdots \\ X_{n1} & X_{n2} \end{pmatrix} \end{matrix}$$

is the evaluation matrix constructed by *n* quality attributes and *two* assessment metrics, in which each X_{ij} represents the measurement results of QA_i by the metric Mj, where i ∈ N, and j ∈ {1,2}. By using these metrics, we guarantee an inclusive assessment by combining the states S1, and S2 in the same matrix. Each horizontal and vertical vector of matrix X is compared to the corresponding optimal vector using the closeness metrics. LAFA depends on these metrics to output the final assessment report.

6.2.3.2 Assessing quality using Euclidean distance as a closeness measure

Definition 7: Euclidean distance between two vectors (O'Neill, 2014)

Let A= (A1, A2, … An), and B= (B_1, B_2, … B_n) be two non-zero real vectors, then

$$d(A,B) = \|A - B\| = \sqrt{(A_1 - B_1)^2 + (A_2 - B_2)^2 + \cdots + (A_n - B_n)^2} \qquad (17)$$

is the Euclidean distance between the vector A and the vector B.

Definition 8: The closeness measure

The closeness measure (metric) denoted by ***clos (A, B)*** is the Euclidean distance d (A, B) between the vectors A and B. The closer the value of *clos (A, B)* is to 0 the closer the vector A is to B. Additionally, the metric *clos* is a judgment of magnitude and not of orientation, which means that the orientation of the vectors A and B does not matter in calculating *clos (A, B).*

MS-QuAAF defines two types of closeness measure, *horizontal* closeness, and *vertical* closeness. In the former, each row of the matrix is compared to the optimal vector. In the latter, we calculate the closeness between each column of the matrix and the optimal vector.

Definition 9: The horizontal closeness metric (**ClosH**)

ClosH consists of calculating the Euclidean distance between each row of the matrix X and the corresponding optimal vector. More accurately, each alternative represents a vector that contains QuARF and RSI assessment values. ClosH permits post-evaluating each quality attribute by comparing these vectors' values to the optimal assessment values by performing the following steps.

Step 1. *Calculating the closeness of alternatives*

Let X= $(Xij)_{nx2}$ be the evaluation matrix, H ={H_1, H_2,…, H_n} be the set of optimal vectors in which, each H_i= $\{\max_{i \in n} QuARF_{QAi}, \max_{i \in n} RSI_{QAi}\}$={1, $\max_{i \in n} RSI_{QAi}\}$, then the closeness between each alternative QA_i and the corresponding H_i is calculated by Eq. (17) as follows.

$$ClosH_{QAi} = \sqrt{(H_{QuARF} - X_{i1})^2 + (H_{RSI} - X_{i2})^2} \qquad (18)$$

Step 2. *Results interpretation*

Calculating ClosH permits judging software architecture as a whole by including evaluation results of design and implementation stages of each QA_i in the final evaluation model (matrix X). The closer the value of ClosH is to 0, the better is the satisfaction of the quality attribute QA_i. The obtained value can lead to two main scenarios:

- $ClosH_{QAi}$ is close to 0: this scenario means that the design rules and NFR responsibilities are both satisfied.
- $ClosH_{QAi}$ is considerably greater than 0: this scenario can be interpreted as follows.

- Design rules are not fulfilled, which may affect NFR responsibilities assigned to QA_i negatively.
- Design rules are fulfilled contrary to NFR responsibilities, which can be interpreted as a partial architecture deviation that concerns only the architecture facet AF_i associated with QA_i.

Other scenarios and interpretations can be emerged according to the value of $ClosH_{QAi}$.

Definition 10: The vertical closeness metric (**ClosV**)

ClosV consists of calculating the Euclidean distance between each column of the matrix X and the corresponding optimal vector. Each column represents a vector that contains the assessment values of all quality attributes of one architecture state. For instance, the first column contains the measurement values of all quality attributes using the metric QuARF. Contrary to ClosH, ClosV permits evaluating the overall quality through each metric M_i against the optimal assessment values by performing the following steps.

Step 1. *Calculating the closeness of attributes (columns)*

Let $X= (Xij)_{nx2}$ be the evaluation matrix, $V= \{V_{QuARF}, V_{RSI}\}$ be the set of optimal vectors in which, $V_{QuARF}= \{max\ QuARF_{QA1},\ max\ QuARF_{QA2},\ \ldots,\ max\ QuARF_{QAn}\}= \{1,1,\ldots 1\}$, and $V_{RSI}= \{max\ RSI_{QA1}, max\ RSI_{QA2}, \ldots, max\ RSI_{QAn}\}$, then the closeness between each column M_j and the corresponding V_j is calculated by Eq. (17) as follows.

$$ClosV_{M_j} = \sqrt{(V_{j1} - X_{j1})^2 + (V_{j2} - X_{j2})^2 + \cdots + (V_{jn} - X_{jn})^2} \qquad (19)$$

Step 2. *Results interpretation*

Calculating ClosV permits measuring the overall architecture quality in a separate way. More specifically, we measure the overall quality (all QAs) at the design stage using the metrics QuARF, and we measure it at the implementation stage using the metric RSI. The obtained value of ClosV can lead to the following scenarios:

- $ClosV_{QuARF}$ and $ClosV_{RSI}$ are close to 0: the overall quality at both development stages is satisfied. The opposite scenario emerges when ClosV is remarkably greater than 0.
- $ClosV_{QuARF}$ is close to 0 and $ClosV_{RSI}$ is not close to 0: this scenario means that the implemented architecture does not meet the designed one, which can be interpreted as an overall architecture deviation. This type of deviation is called architecture erosion (de Silva & Balasubramaniam, 2012).

- $ClosV_{QuARF}$ and $ClosV_{RSI}$ are not close to 0: this scenario represents a total architecture defectiveness in which, both the designed and implemented architecture have deviated from the predefined specifications.
- $ClosV_{QuARF}$ is not close to 0 and $ClosV_{RSI}$ is close to 0: this scenario occurs when the detected design flaws (non-fulfilled rules) are fixed before proceeding with the implementation stage.

Other intermediate values can produce other scenarios. It is up to evaluators to interpret the emerged scenarios.

LAFA uses ClosH and ClosV to conclude the evaluation effort by producing the final assessment report according to the emerged evaluation scenarios. Architects and evaluators can approve or disapprove deviations calculated by those closeness measures. Accepted deviations can be addressed and treated in the next system releases. In the next chapter, we will conduct an empirical study to check the usefulness of the defined metrics suite (Table 6.3) in evaluating architecture quality.

Metrics	Services	Measurement scope
QuARF	RDA	One quality attribute at a time (state 1)
QuARD	RDA	One quality attribute at a time (state 1)
OVF	RDA	Overall at state 1
OVD	RDA	Overall at state 1
RSI	RTA	One quality attribute at a time (state 2)
ClosH	LAFA	One quality attribute at a time (state 1, and state 2)
ClosV	LAFA	Overall at state 1, and state 2

Table 6.3. MS-QuAAF metrics suite

6.3 Conclusion

In this chapter, we have depicted exclusively and in detail, the quality evaluator module and the evaluation process performed through it. Once architecture facets are extracted using the facet projector module, they will be sent directly to the quality evaluator to be assessed. The latter consists of three services that cover the evaluation of architecture at two main development stages, which are the conceptual design and implementation stages. More specifically, we have provided the theoretical aspect of these services by presenting all the mathematical equations used to calculate the evaluation metrics defined within each service. These metrics are generic because they can be applied to assess any quality attribute on the condition that the architectural decisions taken to promote these attributes are specified at the early development stages and placed in the NFR catalog.

The first service called RDA is dedicated to analyzing and gauging the defectiveness of the designed architecture against the design rules specified at the early stages of the development process. This service provides four metrics to calculate design-time architectural defects that represent the parts of the design that do not comply with architectural decisions taken earlier

and specified at the meta-model level. These defects are calculated at the facet level for each involved quality attribute and subsequently, the overall defect of the whole architecture is deduced. We have attributed weights to rules to obtain more accurate evaluation results. The larger the weight of a rule, the higher the severity of its defectiveness and vice versa.

The second service called RTA is dedicated to assessing the implemented architecture. It provides the metric RSI, which is an indicator used to gauge the satisfaction of NFR responsibilities by applying a responsibility decomposition analysis. The latter consists of creating, analyzing, and assessing quantitatively a responsibilities tree called the Responsibilities Satisfaction Tree (RST). We have introduced also weights on the leaf nodes (tactics nodes) of RST to boost up the accuracy of RSI and the equity between the involved responsibilities. The larger the weight of a tactic, the higher its impact on promoting the corresponding quality attribute. The outcome of assessing NFR-responsibilities through RST is to detect non-fulfilled tactics, which would help architects in sanitizing the architecture before deploying the system. However, it is very complicated and costly to apply major changes to architecture at this advanced development stage.

The third service called LAFA is dedicated to drawing conclusions on the architecture quality by analyzing the scenarios that can emerge after calculating the metrics ClosH and ClosV. Finally, a global assessment report is generated accordingly.

In the next chapter, we will provide an implementation of these services by conducting an experimental evaluation through two case studies.

Chapter 7

An experimental evaluation of the framework MS-QuAAF: A case study

In the previous chapter, we have presented the evaluations services defined within the quality evaluator module. More specifically, we have presented the theoretical aspect of this module by providing the mathematical definition for each evaluation metric through a set of equations. In this chapter, we will provide a concrete implementation of each evaluation service through an experimental evaluation, in which two case studies will be treated and discussed. The implementation will illustrate how to use the proposed metrics to assess software architecture during the development process. First, we will start by providing the research methodology followed to answer the research questions asked at the beginning of this dissertation (the introduction). Second, we will present the target architecture that will be assessed during the experimental evaluation. Third, we will present the experiment process, which includes architecture specification, design-time evaluation, NFR responsibilities evaluation, and the final assessment report. Finally, the answers to the research questions will be provided. Additionally, we will present a prototype of the MS-QuAAF tool used to assist evaluators during the assessment process.

Contents

7.1 The experimental evaluation protocol 135
7.2 The Experimental evaluation process 142
7.3 Answering the research questions 162
7.4 Threats to validity 163
7.5 Conclusion 164

7.1 The experimental evaluation protocol

The main goal of performing an experimental evaluation is to answer the research questions asked earlier in this dissertation. These questions address many concerns, such as the ability of the framework to assess software architectures and the effectiveness and accuracy of the proposed metrics. At the beginning of this section, we evoke the research question asked before as follows.

- **RQ1:** Is the proposed framework capable of calculating the defectiveness of the designed architecture?

This question aims to evaluate the effectiveness of the metrics QuARF (Quality Attribute Rules Fulfillment), QuARD (Quality Attribute Rules Defectiveness), OVF (Overall Fulfillment), and OVD (Overall Defectiveness).

- **RQ2:** Is the proposed framework capable of estimating the satisfaction of NFR responsibilities of the implemented architecture?

This question aims to evaluate the effectiveness of the metric RSI (Responsibilities Satisfaction Indicator).

- **RQ3:** Are the proposed metrics capable of deducing architecture deviations?

This question aims to evaluate the effectiveness of the metrics ClosH (horizontal closeness) and ClosV (Vertical closeness) in assessing quality and detecting architecture deviations.

- **RQ4:** Does the proposed framework help in enhancing architecture quality?

This question aims to evaluate the usefulness and the effectiveness of MS-QuAAF in enhancing the architecture quality at the design and implementation stages.

However, before putting the framework under assessment, we have defined a must-follow protocol that determines how we should run the experimental evaluation. The protocol includes the following steps:

a. Selecting the research methodology: In this step, we have selected the research methodology that will be used during the experimental evaluation (case study, experiment, survey, etc.).
b. Selecting target architectures: In this step, we have selected the software architectures that will be assessed using MS-QuAAF's services.
c. Configuring the groups involved in the experiment: In this step, we have chosen the staff that will be involved during the evaluation (designers and developers).
d. Launching the experiment: In this step, we have started the experimental evaluation.

7.1.1 The research methodology

The research methodology followed during the experimental evaluation to answer the above questions is a case study. This research methodology can provide us with research results from real-world projects, which would be difficult to achieve with other research methodologies (Runeson et al., 2012). Moreover, case studies allow finding out what happens, especially within a specific environment.

7.1.1.1 Research purpose

Different research purposes are served by different research strategies including case studies. Three main research purposes can be distinguished: *explanatory*, *exploratory*, and *descriptive* (Runeson & Höst, 2009; Wohlin & Aurum, 2015). Explanatory research is applied to seeking explanations for a problem, mostly in a form of causal relationships. Exploratory research is

applied to determining what is happening, searching for new insights, and generating ideas and hypotheses. Descriptive research aims to portray the status of a phenomenon or situation. According to this classification, our case study falls into the explanatory category. The defined research questions tend to explain and examine the (causal) relationships between the data collected from the experimental evaluation (number of violated rules, weights, NFR responsibilities satisfaction, etc.) and quality attributes satisfaction.

The first RQ examines the relation between design rules (tactics, constraints, etc.) violation, the weight of these rules, and architecture defectiveness rates calculated through the metrics QuARF, QuARD, OVF, and OVD. The statistical analysis must determine the ability and effectiveness of these metrics in calculating the defectiveness of the designed architecture. It displays also the causal relation between rules infringements and the calculated defectiveness by the dedicated service and metrics.

The second RQ examines the relationship between NFR responsibilities, their weights, their satisfaction degrees, and the satisfaction of the associated quality attributes. The statistical analysis must display the causal relationship between NFR responsibilities fulfillment calculated through the metric RSI and quality attributes achievements. It determines also the ability of the metric RSI to calculate NFR responsibilities satisfaction of the realized (implemented) architecture.

The third RQ examines the relationship between the previously obtained results at design and implementation stages, and architecture deviation using the metrics ClosH and ClosV. The statistical analysis must display the relation between these results and deviations. It allows also determining the effectiveness of these metrics in assessing quality and detecting architecture deviations.

The fourth RQ allows examining the relationship between the proposed metrics within their services and architecture quality improvement.

7.1.1.2 Case study design

After we had formulated research questions, we adhered to the protocol of the case study design defined in (Perry et al., 2004; Runeson et al., 2012). According to this protocol, the design of a case study can be divided into three steps: planning, data collection, and data analysis.

a. *Planning*. In this phase, we have defined the case (the phenomenon in its real context), fixed its objective (explanatory), and selected the method used to collect data and evidence. In this context, data will be collected from the experimental evaluation performed using MS-QuAAF. Additionally, we have planned and selected the type of case study that we will conduct in this experimental evaluation. There are four known types of case studies (Perry et al., 2004): holistic single-case, embedded single-case, holistic multiple-case, and embedded multiple-case. Our case study belongs to the embedded single-case category, in

which two analysis units are included (figure 7.1). The first unit is the architecture of an auction web application (SA1), whereas the second unit is the architecture of a desktop data visualization tool

b. *Data collection.* According to (Wohlin & Aurum, 2015), archival research, surveys, experiments, and simulation are the commonly used quantitative data collection methods. In this dissertation, data are mainly collected from the experimental evaluation. Among the collected data, we can find the number of macro-groups, the number of rules within each macro-group, the number of violated rules in each macro-group, weights of macro-groups, the satisfaction score attributed to each tactic node of the RST (Responsibilities Satisfaction Tree), and the weight of each tactic node.

c. *Data analysis.* The data analysis method conducted in this dissertation tends to be statistical. All the equations defined in chapter 6 are used to treat and analyze the previously collected data. In the design stage, the defectiveness rates from the macro-group extent to the overall extent are calculated and interpreted. In the implementation stage, the satisfaction degree of NFR responsibilities associated with each quality attribute is calculated and interpreted. The obtained results in both development stages are also used for the final assessment report through an evaluation matrix.

All the interpretations in this phase aim to be causal (sections 7.2.2, 7.2.3, and 7.2.4). The obtained results through metrics at both development stages are related directly to quality attributes achievement and architecture deviations. For instance, high architecture defectiveness (QuARF is very close to zero) may cause high degradation of the associated quality attribute, or violating rules rated as extremely or strongly important (associated with high weights) may cause the increase of architecture defectiveness.

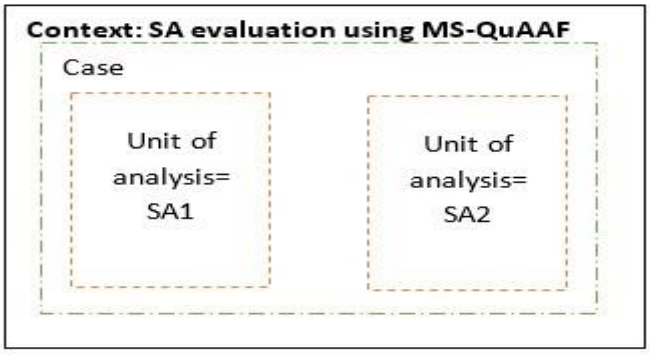

Figure 7.1. The structure of our embedded single-case study

7.1.2 Target architecture

In this study, we have proposed two academic software architectures to evaluate MS-QuAAF. This choice is constrained by the fact that specification, design, and implementation of industrial real-world projects are very hard to obtain due to the restricted policy of their owners and copyright protection. Furthermore, we wanted to get involved in the development process from the beginning to evaluate the effectiveness of the proposed metrics in assessing and enhancing the architecture quality throughout this process. The first proposed architecture

(SA1) concerns an auction web application that manages online auctions. The second architecture (SA2) concerns a data visualization desktop tool. These applications can be described as follows.

- The auction application (figure 7.2 (a)) allows bidders to search, select, and purchase bids. Its architecture is organized around five components: the J2EE server, the client, the broker, the client-side proxy, and the server-side proxy. The broker represents an intermediary that attempts to search the appropriate server to satisfy clients' requests. A client cannot connect to a server without passing by the broker. This separation allows replacing failed servers dynamically. Proxies are used to relate clients and servers to the broker. The most important component in this system is the J2EE server. The latter is architecturally organized as a set of layers or tiers, which makes it a multi-tier server. If a bidder executes an action (e.g., purchasing), a message will be sent from the top layer to the bottom ones. The first layer (the presentation layer) comprises java Servlets that would allow generating dynamically the user interface to log into the system to execute the desired actions. The second layer (the business layer) encompasses the business components, which are the Enterprise Java Beans (EJB). The third layer (data access layer) provides services (e.g., Java Database Connectivity) that would allow the connection to the auction database and execute the desired queries.
- The visualization application (figure 7.2 (b)) is a tool that allows reading data sources, such as databases and CSV files as input and displaying the chosen graphical representation (charts, graphs…etc.) as output. The architecture of this tool follows a layered architecture pattern, in which there are exactly three layers. The top layer permits converting data sources into temporary data models. The second layer contains adapters that convert the temporary models, such as Excel and database models into a generic data model. The third layer contains different types of charts and graphs used to represent the generic data model visually. However, to access this layer, the generic data model of the second layer should connect to the façade component to use the desired graphical elements (to reduce complexity), which means that the third layer adopts the façade architectural pattern.

7.1.3 Groups configuration

According to the number and size of the projects under-experiment, we have formed and supervised two development groups of master's degree students (Table 7.1). These students are trainees in a Startup software development company called *SDI consulting*[1]. The first group is assigned to the auction application project, whereas the second one is assigned to the data visualization project. The first group includes two designers with short experience in Alloy (Jackson, 2012, 2019) language, and three java programmers. On the other hand, the second group comprises two moderate OCL (Object Constraint Language) (*About the Object*

[1] Société de Développent Informatique

Constraint Language Specification Version 2.4, n.d.) designers and two Java programmers. We provided each group with the corresponding architecture specification that includes functional and non-functional requirements. Accordingly, designers from each group have proceeded with the design process relying on these specifications (meta-models). More specifically, the yielded design artifacts for both projects consist of a set of UML diagrams, particularly the component and class diagrams. After the design stage has been completely accomplished, we have assessed the produced design through the service RDA. Subsequently, designers have coordinated with programmers to explain to them the design and the architectural constraints (rules) made to promote the specified quality attributes. By finishing this stage, the mission of both teams was considered accomplished, and the second phase of evaluation has started by calling RTA and LAFA, respectively.

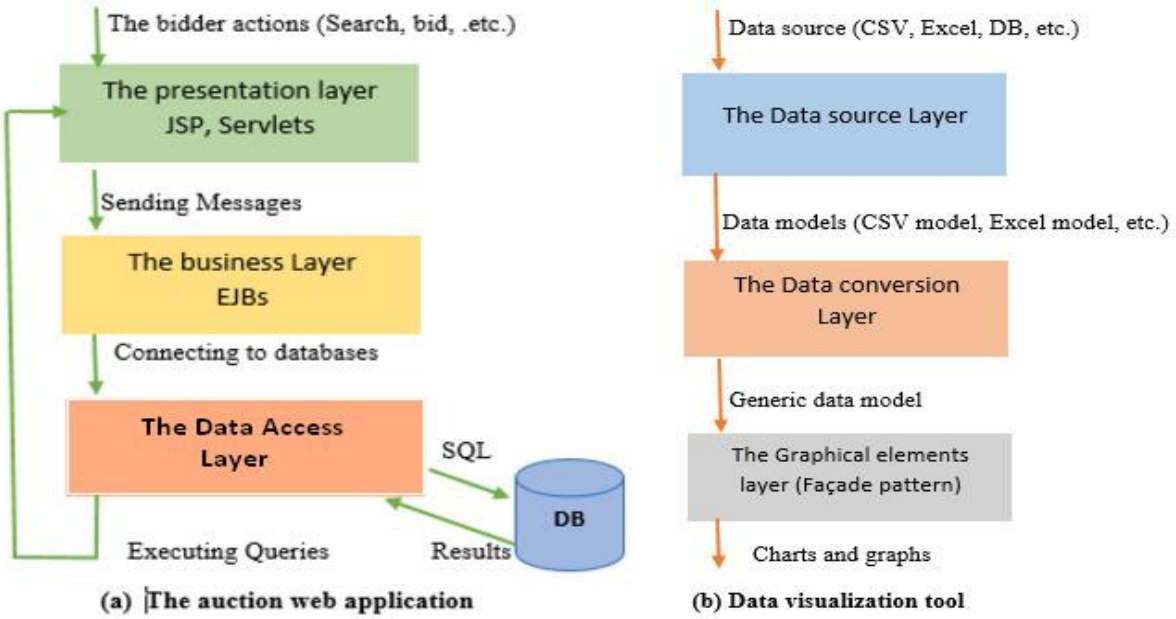

Figure 7.2. A conceptual representation of the target architectures SA1 and SA2

	Group 1	**Group 2**
# Members	5 students	4 students
Project	The auction application (SA1)	The data visualization application (SA2)
Design skills	Alloy, UML	OCL, UML
Programming skills	J2EE (Jakarta EE)	Java
# Designers	2 students	2 students
# Programmers	3 students	2 students

Table 7.1. Development groups' configuration

7.1.4 The MS-QuAAF tool prototype

In order to facilitate, expedite, and automatize the evaluation process through MS-QuAAF, it would be better to develop a tool that implements the architecture and the philosophy of the framework. In this dissertation, we have developed a prototype tool based on the architecture of MS-QuAAF. More specifically, the tool encompasses two main modules: the facet projector and the quality evaluator (figure 5.1). The first module implements the concept of facet projection to assist architects in extracting facets from large meta-models. The second module

implements the evaluation services RDA, RTA, and LAFA to calculate the proposed evaluation metrics.

Figure 7.3 shows the main window of the MS-QuAAF tool. This window provides a simple and intuitive flat design that encompasses a set of buttons from which we can launch all the evaluation services. The first row of these buttons displays three child windows, each of which allows starting and exploiting the services RDA, RTA, and LAFA, respectively. In the second row, the button called *Facets* is dedicated to extracting architecture facets. In this prototype, the facet projector supports only the language Alloy; however, it is easy to add other specification languages by providing the corresponding implementation.

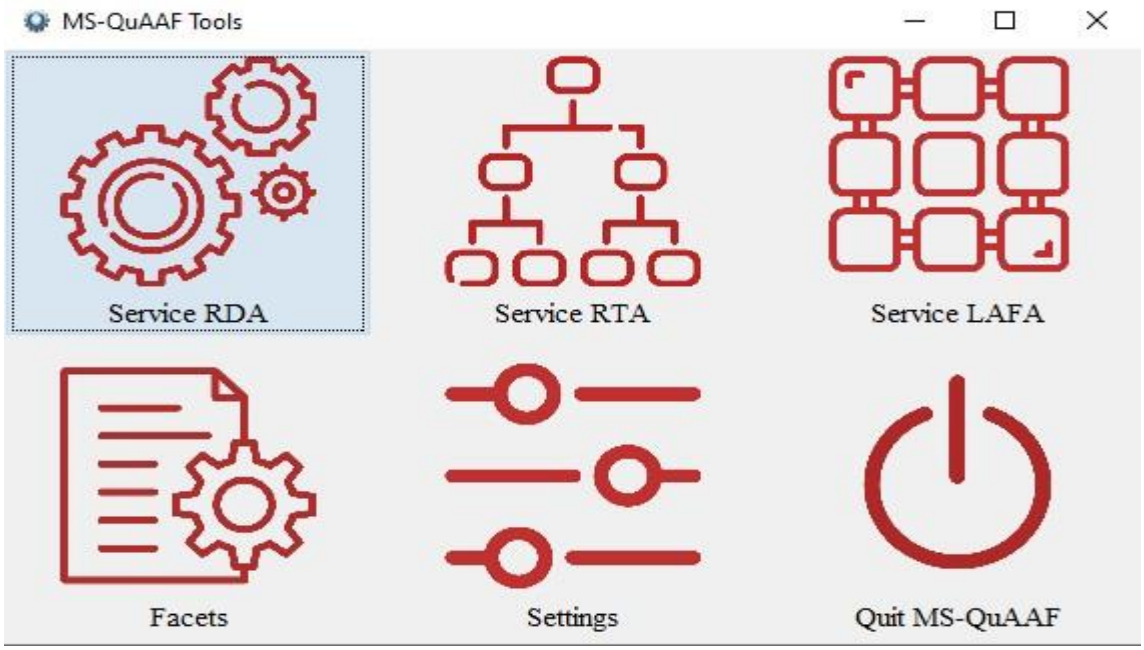

Figure 7.3. The main window of the MS-QuAAF tool prototype

The tool should connect to the NFR catalog to extract data, such as the architecture name, stakeholders' quality attributes, and design rules. Moreover, the evaluation results obtained through metrics are stored in this catalog. The latter can be defined and implemented using many techniques, such as databases, XML, or any other techniques that best fit the architects' needs. In our prototype, we have used a relational database and XML technology to manipulate architectures' data. Figure 7.4 shows an Entity/Relation diagram that portrays the main entities of the NFR catalog:

- Architecture: This entity represents the architecture under assessment.
- Quality attribute: This entity represents the target quality attributes. The obtained evaluation results (QuARF, QuARD, etc.) are stored within this entity.
- Macro-group: This entity depicts the macro-groups that will be used by the service RDA.
- Rule: This entity encompasses design rules that will be analyzed by the framework.

Furthermore, we have used XML, particularly with the second service RTA where RST trees are mapped to XML documents to facilitate the process of calculating the metric RSI. It should be noted that it is possible to implement the NFR catalog fully in XML; however, using relational databases to manage some parts of the catalog (particularly data used by RDA) can provide us total control on data, and thus avoiding flaws and anomalies (e.g., rules redundancy). Examples of using the tool will be depicted in the rest of this chapter.

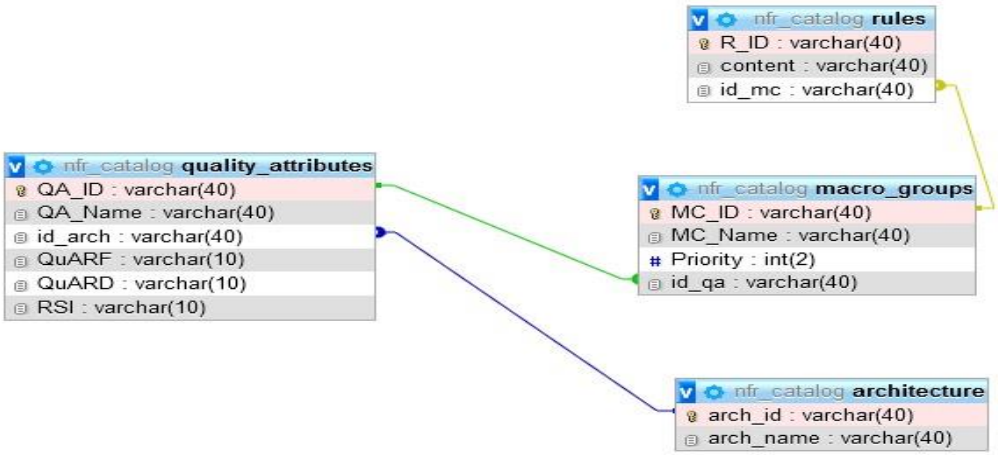

Figure 7.4. An Entity/Relation diagram of the NFR catalog

7.2 The Experimental evaluation process

The experimental evaluation can be divided into four major phases, which are the architecture specification, the design-time evaluation, the implementation-time evaluation, and the final assessment report.

7.2.1 Architecture specification

Architecture specification is the cornerstone on which the proposed evaluation is based. According to the target quality attributes of each project, we have specified SA1 and SA2 in a detailed manner taking into account the design skills of each group. The specification permits the construction of the NFR catalog for each architecture. The selection of quality attributes is motivated by the business goals of these projects. Table 7.2 depicts the quality attributes and the specification language for both projects as follows.

7.2.1.1 Specifying SA1 using Alloy

Alloy is a general-purpose first-order specification language that uses predicates and facts to specify systems. We have used Alloy to specify SA1 taking into account the desired quality attributes, which are maintainability, performance, and availability. The specification consists of defining a set of rules to promote each one of these attributes (70 rules in total). For instance, to promote the server's maintainability, the main architecture decision made is to adopt the J2EE architecture, which follows the multi-tier architectural style. To promote availability, we defined rules that manage the communication between clients, servers, and the broker component. Listing 7.1 shows an excerpt of the Alloy specification of SA1, which depicts the following elements (see **Appendix A**).

- The main entities of the system are expressed as Alloy *Signatures*, such as layers, EJB components, database components, broker components, and Java Servlets
- The relationships between these components are expressed as Alloy *relations*.
- Design rules, which are expressed as Alloy *facts*, *predicates*, and within *signatures* declaration. These rules should keep the system consistent by managing and constraining the relationships: component-component, component-layer, and layer-layer.

For instance, to promote maintainability, we have defined some rules as follows.

- Adopting the multi-tier J2EE architecture as the main architectural style of the server.
- The architecture is divided into three layers (tiers).
- The first tier (the lower layer) should contain only the components that interact with the auction database to execute clients' queries.
- The second tier should contain only the business components (EJBs).
- The third tier (the top layer) should contain only the presentation components, which are Java Servlets.
- Messages can only be sent from the third layer to the second layer, from the second layer to the first layer, and from the first layer to the third one.
- The degree of coupling between two components should not exceed 5.
- If the degree of coupling exceeds 5 then insert an intermediary component.

To promote performance, we have specified some rules as follows.

- The number of layers should not exceed five. A high number of layers will decrease performance (Bass et al., 2012).
- Use Java DataSource instead of the DriverManager to connect to the database.
- Use connection pooling to data sources to boost up performance.
- Use PreparedStatment instead of ordinary statements.
- Use batched SQL queries.

Most rules, particularly the main rules that manage the structure of the system and the relationships between layers and components were specified with Alloy. However, some behavior rules that necessitate a considerable Alloy specification effort were specified using natural language in order to reduce the complexity of the specified model and to facilitate the design of the system (e.g., improving bandwidth).

	SA1	**SA2**
Quality attributes	Maintainability, performance, availability	Maintainability, extensibility
# Rules	70	45
Specification language	Alloy	OCL

Table 7.2. Quality attribute, number of rules, and specification language of SA1 and SA2

7.2.1.2 Specifying SA2 using OCL and eclipse Ecore

OCL is a pure formal specification language that can be used to specify application-specific constraints. We have used OCL constraints to specify SA2 according to the desired quality attributes, which are maintainability and extensibility (45 rules). Practically, we have used The OcLineEcore plugin (figure 7.5) that allows enriching Ecore and UML meta-models with OCL constraints inside the Eclipse Modeling Framework (EMF). Accordingly, we have embedded OCL constraints (invariants) within the Ecore meta-model that describes the architecture SA2 (see **Appendix B**). We have found that adopting this embedded specification philosophy allows enriching meta-models and thus enhancing understandability and readability unlike using separate OCL specifications. In this context, we have not used separate OCL specifications because it requires an extra effort to understand the final meta-model and the mapping to the OCL constraints.

Listing 7.2 shows the specification of the SA2 meta-model and the embedded OCL constraints. This specification depicts the main classes of the system, which are

- Architectural layers: Architecture, Reader_Layer, Converter_Layer, and Renderer_Layer. These layers are expressed as classes, each of which contains attributes and operations (methods).
- Architectural components: Data_Reader, Data_Converter, and Data_Renderer. These components are also expressed as classes, each of which is encompassed within the corresponding layer.
- Interfaces: DataAdapter and Façade. These interfaces should be implemented by the above elements.
- OCL constraints: Constraints are mostly expressed as OCL *invariants*. These constraints are embedded within the system's classes, interfaces, and their declared operations bodies.

For example, to promote maintainability, the following subset of OCL invariants is specified (some rules are similar to the maintainability rules of SA1).

- Follow the layered architectural style as the primary architectural pattern of SA2, in which the number of layers is exactly three (it differs a little bit from the multi-tier style defined in SA1, particularly in message circulation).
- The first layer should contain data readers, such as CSV, MS Excel, and databases readers.
- The second layer should contain data converters
- Data converters should transform inputted data to a standardized format (tabular format) specified by the interface DataAdapter.
- The third layer should encompass rendering components charged with displaying data in diverse graphical formats (e.g., charts).
- The third layer should adhere to the Façade pattern specified by the interface Façade.

- The interface Façade is responsible for granting access to the third layer and selecting the appropriate rendering components subsequently.
- The first layer should not receive any message, whereas the third layer is not allowed to send any message (unlike SA1).

To promote extensibility (Bass et al., 2005), we have defined rules that allow adding new components and functionalities without affecting the system design. Accordingly, we have defined rules that allow the system to be extensible without affecting the layered style:

- To add a new rendering type, it suffices to add the corresponding rendering component to the third layer and re-implementing the façade interface.
- To add a new converter, it suffices to add the corresponding converter component to the third layer.
- To support a new data format, it suffices to add a new reader to the first layer.
- Adding new components will not affect the messages sending and receiving paradigm.
- A layer is allowed to add new components.
- Adding new layers is allowed without exceeding five layers.
- To add or modify data format, it suffices to re-implement the interface DataAdapter.

However, we have designed the system to be extensible to a precise extent. Making the system extensible to a large extent may affect maintainability and harden the system maintenance. Therefore, we have preferred maintainability at the expense of extensibility (tradeoff between maintainability and extensibility).

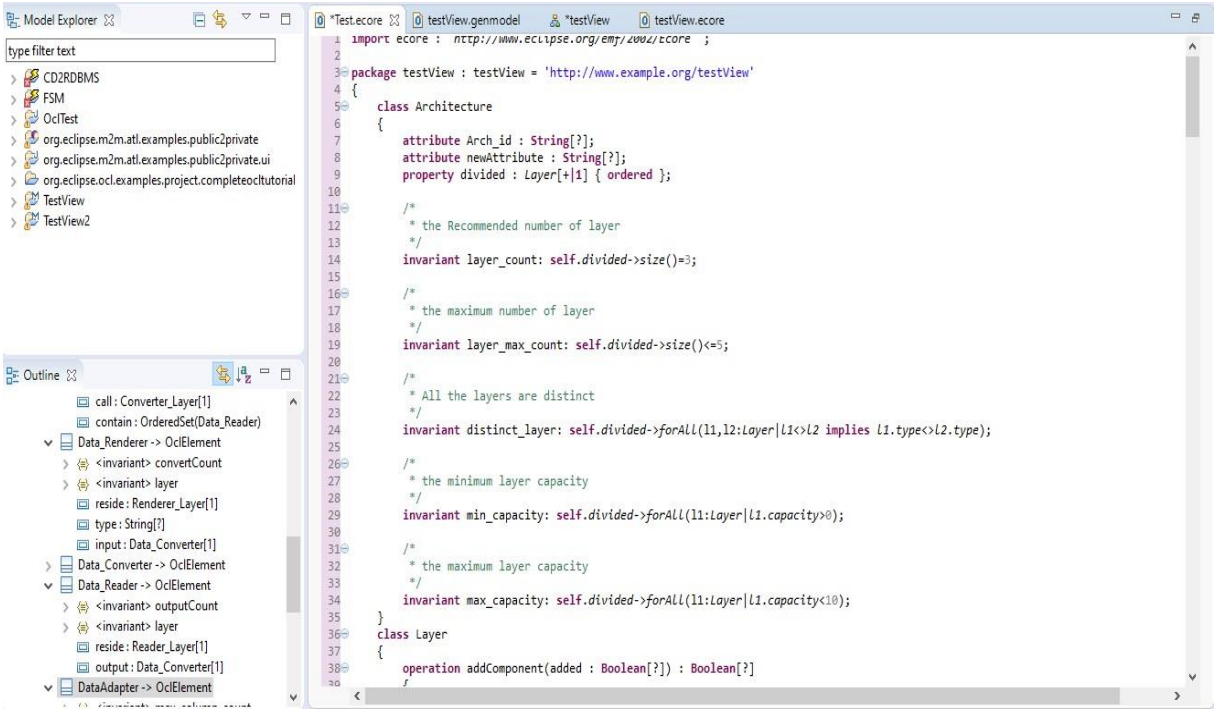

Figure 7.5. The OCLinEcore plugin used to embed OCL invariants within EMF.

```
1  module J2EE_Server4
3  abstract sig Component {coupledTo:Component}
4  sig intermediaryComponent extends Component {}
5  abstract sig layer {contains: some Component, message:one layer}
6  //sig intermediaryLayer extends layer {}
7  abstract sig Servlet_auction extends Component{calls:Ejb_auction}
8  abstract sig Ejb_auction extends Component{}
9  abstract sig Service_auction extends Component{}
10 one sig Server{divided_into:some layer, deliver:Response  ->  Broker, addC: set
intermediaryComponent}
...
16 ///////Declaring the broker/////////
17 sig Client {send:Request ->Broker}
18 sig Registry {contents: Service}
...
25 ////////////Declaring Layers or Tiers//////////
26 //layer03
27 one sig Presentation_Layer extends layer {}
28 sig login extends Servlet_auction {}
29 sig Account {searches:search_article, buys:buy_article, pays:pay_article}
....
33
34 //Layer02
35 one sig Business_Layer extends layer {}
36 one sig login_Ejb extends Ejb_auction {connect:login_DB}
…
40
41 //Layer01
42 one sig Data_Layer extends layer {}
43 one sig DB extends Service_auction{}
44 sig login_DB extends Service_auction{openAccount:Account}
....
49 ///The presentation layer must contain only Servlet Components
50 fact {all s:Servlet_auction, e:Ejb_auction, r:Service_auction, l:Presentation_Layer |s
in l.contains && e not in l.contains  && r not in l.contains}
51
52 ///The Buisiness layer must contain only EJBs Components
53 fact {all s:Servlet_auction, e:Ejb_auction, r:Service_auction, l:Business_Layer|e in
l.contains && s not in l.contains  && r not in l.contains}
....
58 // the architecture is divided into three layers, which are the presentation, data, and
Business Layers
59 fact {all l:layer, a: Server | l in a.divided_into}
...
64 //Datasource,connection pooling to enhance performance
65 one sig Connection_DB{pooledTo:DataSource, execute:Query}//using pooling to enhance
performance
...
70 fact {all q:Query| q.belongTo=PreparedStatement}
71 fact {all q:Query| q.executedAs=Batch}
72
73 //////messages rules between layers
74 //the presentation layer can send messages to the business layer and not vice versa
75 fact {all p:Presentation_Layer, b:Business_Layer| p.message in b}
…
81 fact {all p:Presentation_Layer, b:Business_Layer,d:Data_Layer|p.message not in p &&
82                                  b.message not in b && d.message not in d}
83
84 /////Coupling between component
85 //coupling between 2 components should be lesser than 8
86 fact{all c:Component | #c.coupledTo <=7}
87 // a component cannot be coupled to it self
88 fact{all c:Component |c not in c.coupledTo}
......
90 fact {all c:Component, c1:intermediaryComponent, s:Server |#c.coupledTo<=7 implies c1
```

Listing 7.1. The Alloy meta-model of SA1

```
1  import ecore : 'http://www.eclipse.org/emf/2002/Ecore' ;
2
3  package testView : testView = 'http://www.example.org/testView'
4  {
5      class Architecture
6      {
7              attribute Arch_id : String[?];
8              attribute newAttribute : String[?];
9              property divided : Layer[+|1] { ordered };
10
11             /*
12              * the recommended number of layers
13              */
14             invariant layer_count: self.divided->size()=3;
15
16             /*
17              * the maximum layer capacity
18              */
19             ....
35     }
36     class Layer
37     {
....
50             operation sendMessage(m : Layer[?]) : Boolean[?]
51             {/* */
52                    body: Layer.allInstances()->forAll(p|if p.oclIsTypeOf(Reader_Layer)
or                  p.oclIsTypeOf(Converter_Layer) then sendMessage(p)
53                            else
54                            not sendMessage(p)
55                            endif);
56             }
....
66     }
67     class Renderer_Layer extends Layer
68     {
....
75             invariant
76             MsgDirection1:
77
78             Layer.allInstances()->forAll(p|if p.oclIsTypeOf(Reader_Layer) or p.
               oclIsTypeOf(Converter_Layer) then
79                    not self.sendMessage(p)
80
81                    else  self.sendMessage(p) endif);
82
....
88
89             Layer.allInstances()->forAll(p|if  p.oclIsTypeOf(Converter_Layer)  then
90                    self.receiveMessage(p)
91
92                    else  not self.receiveMessage(p) endif);
93
....
104            invariant selectRenderer:
105            self.follow.allowAccess() implies self.follow.selectRenderer()->size()=1;
106
....
110            invariant content:
111        self.contain->    forAll(p|      p.oclIsTypeOf(Data_Renderer) );
112    }
113    class Converter_Layer extends Layer
123            let send : Renderer_Layer = self.call in
124            if self.call.oclIsTypeOf(Renderer_Layer)then sendMessage(send)else not
sendMessage(send) endif;
.....
```

Listing 7.2. The OCL embedded specification within the Ecrore meta-model of SA2

In both architecture specifications, rules do not have the same architectural weight. In the NFR catalog, we have assigned to each specified rule an importance level according to a four-level importance scale. Table 7.3 depicts the importance scale used to classify rules according to their contribution to promoting the corresponding quality attributes. This classification facilitated the creation of macro-groups and the calculation of their weights. Hence, violating extremely and strongly important rules will engender a disastrous impact on the intended design and architecture quality. For instance, omitting the broker in SA1 or violating the layered style rules in SA2 may affect severely the availability and the maintainability attributes, respectively. On the other hand, rules at a lower importance level can cause architectural damages and quality degradation to a lesser extent.

In the next phases, we will use the defined metrics to evaluate the quality of the designed and implemented architecture using the rules specified in the current phase.

Importance scale	**Rules example (SA1)**	**Rules example (SA2)**
Extremely important	Clients and server should communicate through the broker (availability)	A layer can only use the services of its adjacent lower layer (maintainability)
Strongly important	The server should follow the J2EE layered architecture (maintainability)	Adding new data sources should be realized by adding data adapters in the second layer (extensibility)
Moderately important	The size of classes and JSP pages should be reduced (performance and maintainability)	Using the façade style in the third layer to reduce complexity (maintainability and extensibility)
Slightly important	Use image caching and data compressing (performance)	Inserting intermediaries to reduce high coupling (maintainability and extensibility)

Table 7.3. Examples of SA1 and SA2's general rules with their importance level

7.2.2 Design-time evaluation

This evaluation phase starts after the accomplishment of the design stage, in which designers of each group should follow the corresponding architecture specification. However, this is not always the case because designers may infringe on some rules, which causes the deviation of the designed architecture from the specified one. In this phase, the evaluation effort is divided into five steps, each of which can be performed automatically or semi-automatically, except step 4 that necessitates architects to analyze and check the design to uncover rules violations.

Step1. *Facets projection*

In this step, we have projected facets from the NFR catalog of each project according to the target quality attributes. We extracted from SA1 three facets, which are AF1, AF2, and AF3

related to maintainability, performance, and availability, respectively (Table 7.6). From SA2, we have extracted two facets, which are AF4, AF5 mapped to maintainability and extensibility, respectively (Table 7.7). Each facet encompasses architectural decisions made to promote the relevant quality attribute. We have used the facet projector of the MS-QuAAF tool to project architectural facets by implementing the projection algorithm (see chapter 5) defined in (Kadri et al., 2020).

The facet projector GUI (Graphical User Interface) provides an easy and straightforward way to extract facets from the overall Alloy meta-models through the following steps (figure 7.6).

a) Loading the Alloy specification that depicts SA1 (.als file) into the left window panel by clicking on the button *Load*.
b) Specifying the projection query in the top window panel. The query was written as "**Select** Component (*), layer (*), Server, Message (*), Account **from** SA1 **into** AF1 **where** quality_attribute =*Maintainability''*. The query designates the elements to be extracted from SA1 according to the projection criterion *Maintainability*. These elements represent Alloy *Signatures* (close to the notion of classes in the object-oriented paradigm), *Facts, and Predicates* mapped to the criterion. The annotation (*) indicates to the facet projector to extract the signature and all its inherited signatures. Each Alloy fact related to the selected signatures is also extracted.
c) Selecting the designated Alloy elements using the associated checkboxes.
d) Creating the facet AF1 by clicking on the button *Create Facet*. The yielded facet is displayed in the right window panel.
e) Saving AF1 by clicking on the button *Save Facet*.

The produced facet is divided into two logical parts (see the example shown in section 5.3.6). The first part represents the declaration of Alloy signatures and the relationships between them. Among these signatures, we can find architectural layers, Servlets, EJBs, etc. The second part of AF1 represents the relevant Alloy facts (rules), such as components that must exist in each layer, the relationships between these components, coupling degree, etc. These facts are supposed to be followed by designers and developers to promote the maintainability attribute.

Step2. *Creating macro-groups*

In this step, we have sliced facets into a set of macro-groups according to the importance scale depicted in table 7.3. As a result, four macro-groups (MG1, MG2, MG3, and MG4) have been created from each extracted facet; in which each macro-group represents a subset of rules. In this context, MG1, MG2, MG3, and MG4 contain the extremely, strongly, moderately, and slightly important rules, respectively. The slicing process was performed automatically according to the importance level assigned to each architectural decision in the NFR catalog.

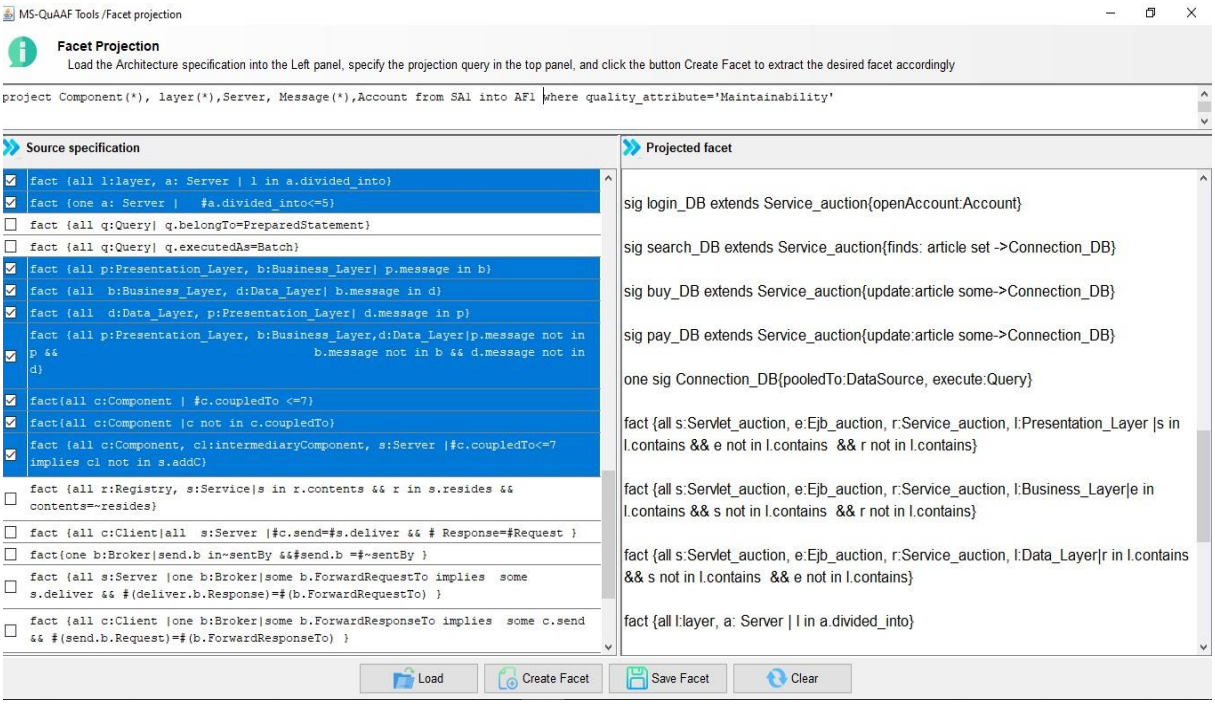

Figure 7.6. Extracting a facet through the Facet projector GUI

Step3. *Calculating macro-groups weight*

Calculating the weight of macro-groups consists of creating the pairwise comparison matrix (Table 7.4) using the eq. (7) and the Saaty scale (Table 6.2). More specifically, the matrix is filled using a pairwise comparison between macro-groups according to the importance levels of the Saaty scale. For instance, MG1 is of strong importance than MG3 (the attributed value is 5), and MG1 is almost extremely important than MG4. Inversely, MG3 and MG4 are compared to MG1.

Table 7.4. The initial Pairwise comparison matrix

	MG1	**MG2**	**MG3**	**MG4**
MG1	1	2.00	5.00	8.00
MG2	0.50	1	3.00	6.00
MG3	0.20	0.33	1	3.00
MG4	0.12	0.17	0.33	1
Sum	**1.82**	**3.5**	**9.33**	**18**

Table 7.5. The normalized comparison matrix

	MG1	**MG2**	**MG3**	**MG4**
MG1	0.54	0.57	0.55	0.45
MG2	0.27	0.28	0.32	0.33
MG3	0.11	0.094	0.1	0.16
MG4	**0.066**	**0.048**	**0.035**	**0.055**

After we had filled the matrix with the comparison values, we normalized this matrix using eq. (8). More specifically, we divided each value by the sum of the corresponding column calculated in table 7.4, which gave us a new matrix (table 7.5). The latter was used to calculate the weight of each macro-group using eq. (9) as follows.

$\mathbf{W_{MG1}}$= (0.54 + 0.57 + 0.55 + 0.45)/4=0.527, $\mathbf{W_{MG2}}$= (0.27 + 0.28 + 0.32 + 0.33)/4= 0.3, $\mathbf{W_{MG3}}$= (0.11 + 0.094 + 0.1 + 0.16)/4=0.118=, and $\mathbf{W_{MG4}}$= (0.066 + 0.048 + 0.035 + 0.055)/4=0.052, where $W_{MG1} + W_{MG2} + W_{MG3} + W_{MG4}=1$.

These weights will be used in the next step to calculate design-time metrics to assess *state 1* of both SA1 and SA2.

Step4. *Calculating QuARF, QuARD, OVF, and OVD*

This step requires architects to perform design reviews to uncover the potential rules infringements. Therefore, we have analyzed the design to verify its conformance to the architecture specification. Subsequently, we have calculated QuARF and QuARD to measure rules fulfillment and defectiveness.

Table 7.6 and Table 7.7 show the measurement results of SA1 and SA2, respectively. The following sub-steps are performed in sequence to obtain these results as follows.

a) Calculating macro-groups fulfillment (MGF): to calculate MGF_i, we have analyzed each macro-group to extract the number of violated rules. For instance, the notation MG1 (8:2) in the third column of table 7.6 and table 7.7 means that the number of MG1's rules (facet AF1) is 8, in which there are 2 rules violated by designers. Therefore, the MGF of MG1 is calculated using eq. (1) and eq. (2) as follows.

   MGF_1= (1+1+1+1+1+1+0+0) *0.527/8≈0.4, where 0.527 is the weight of MG1, and 8 is the total number of rules of MG1. It is to be noted that the optimal value of MGF_1 is equal to 0.527 if all rules are satisfied. Similarly, we calculated MGF_2, MGF_3, and MGF4 of all quality attributes for both projects.

b) Calculating QuARF and QuARD (fourth and fifth columns): QuARF of each quality attribute is calculated by summing the MGF of all macro-groups using eq. (3). For instance, the QuARF and QuARD of maintainability of SA1 are calculated as follows.

   $QuARF_{maintainability}$=0.4+0.17+0.079+0.029=0.68.
   Subsequently, we calculated the maintainability QuARD using eq. (5) as follow.
   $QuARD_{maintainability}$= (1-0.68) *100=32%.
   In the same way, we calculated QuARF and QuARD of all quality attributes for SA1 and SA2.

c) Calculating OVF and OVD: the overall rules fulfillment OVF of each architecture is calculated using eq. (4) as follows.

   OVF_{SA1}= (0.68+0.63+0.79)/3=0.7.
   OVF_{SA2}= (0.77+0.87)/2=0.82.
   Subsequently, the overall defectiveness is calculated using eq. (6) as follows.

OVD_{SA1}= (1-0.7) *100=30 %.

OVD_{SA2}= (1-0.82) *100=18 %.

Quality attributes	Facets	Macro-groups score (#rules: #violated rules)				QuARF	QuARD
Maintainability	AF1	MG1 (8:2)	MG2 (9:4)	MG3 (6:2)	MG4 (7:3)	0,67	33 %
		0,4	0,17	0,079	0,029		
Performance	AF2	MG1 (6:2)	MG2 (7:3)	MG3 (5:2)	MG4 (4:1)	0,63	37 %
		0,35	0,17	0,07	0,038		
Availability	AF3	MG1 (5:0)	MG2 (4:2)	MG3 (5:2)	MG4 (5:1)	0,79	21 %
		0,527	0,15	0,07	0,04		

Table 7.6. Calculating QuARF and QuARD of SA1

Quality attributes	Facets	Macro-groups score (#rules: #violated rules)				QuARF	QuARD
Maintainability	AF4	MG1 (6:1)	MG2 (7:2)	MG3 (7:3)	MG4 (5:1)	0,77	23 %
		0,44	0,22	0,067	0,041		
Extensibility	AF5	MG1 (3:0)	MG2 (7:2)	MG3 (5:2)	MG4 (4:0)	0,87	13 %
		0,527	0,22	0,071	0,051		

Table 7.7. Calculating QuARF and QuARD of SA2

Using the MS-QuAAF tool

The MS-QuAAF tool made the above steps of calculating architecture defectiveness transparent to the user because QuARF and QuARD are calculated automatically by one button click. It suffices to follow two simple steps to obtain the desired results. First, the user is invited to select the quality attribute under assessment from a list of stakeholders' attributes. This list is obtained automatically by interrogating the NFR catalog (figure 7.7). Second, by clicking the button *next*, a second page that contains macro-groups and their rules will be displayed to the user (figure 7.8). The latter should select the satisfied rules within each macro-group, and then click the button calculate to display the obtained results.

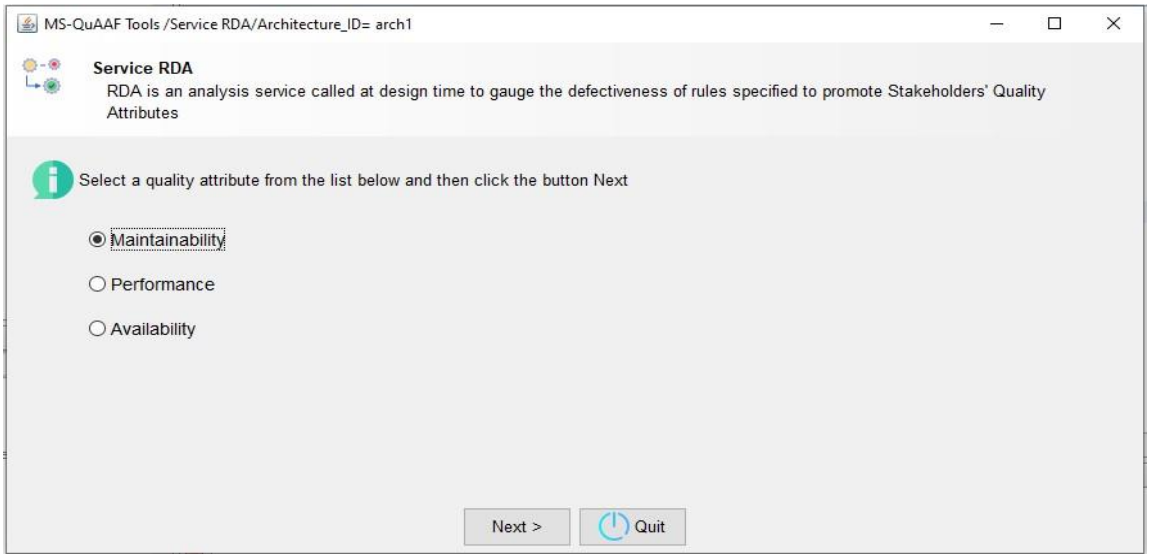

Figure 7.7. Selecting the target quality attribute

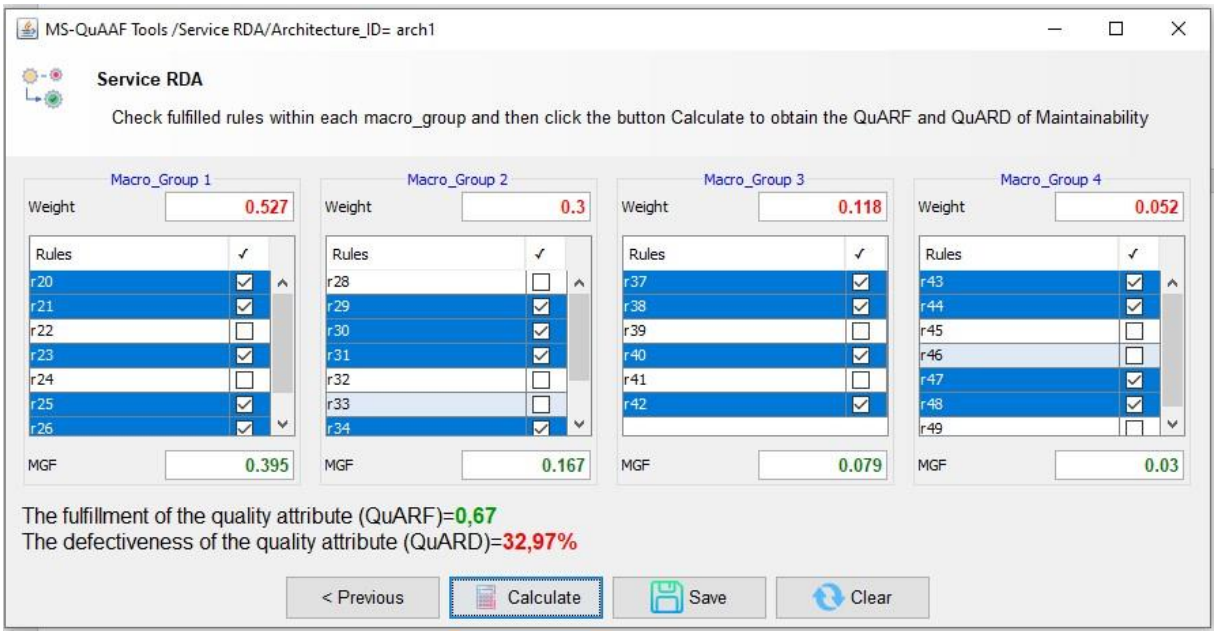

Figure 7.8. Calculating QuARF and QuARD using the MS-QuAAF tool

Subsequently, the obtained results can be saved in the NFR catalog by clicking the button *Save*.

Step 5. *Results interpretation and discussion*

Calculating QuARF and QuARD allowed us to judge both architectures after the completion of the design stage. Figure 7.9 reflects the obtained results of the maintainability and extensibility attributes in the form of a double ring representation. The external ring represents macro-groups, in which the size of sectors reflects the importance level of the involved macro-groups. The internal ring represents the defectiveness (red zones) and fulfillment (green zones) rate of each macro-group. More specifically, figure 7.9 (a) shows that the facet AF1 mapped to the maintainability attribute suffers from many architectural defects, which increased the defectiveness rate up to 33%. Mainly, this is due to the violation of some rules rated as extremely and strongly important. Besides, the infringement of MG3 and MG4's rules caused some damages but to a limited extent. On the other hand, figure 7.9 (b) shows that the extensibility attribute (AF5) has fewer defects, in which the defectiveness is equal to 13 %. This is the consequence of satisfying all extremely important rules and most of the other rules. For the other attributes, performance (AF2) is the most defective one (37%); however, availability (AF3) and maintainability (AF4) have lesser defects (21% and 23%, respectively) because most of their high-prioritized rules are satisfied.

The defects that affected the quality attributes of SA1 increased the overall architecture defectiveness to 30% (Figure 7.10 (a)). This is the result of the deviation from the rules specified earlier, especially the highly important ones. At the same time, the OVD of SA2 is lesser (18%) because the conformance to the architectural specifications is higher than SA1 (Figure 7.10 (b)).

Calculating defectiveness and detecting deviations at this stage permit fixing flaws before proceeding with the implementation stage. However, the accuracy of the obtained results depends on the quality of architecture specification, the ability to determine correctly the importance level of the specified rules, and the correctness of performing rules analysis.

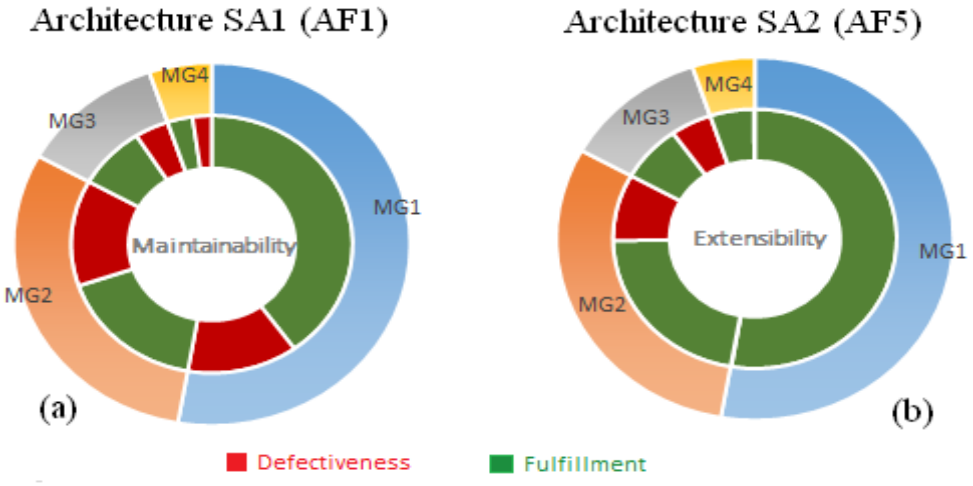

Figure 7.9. A graphical representation of architecture defectiveness of AF1 and AF5

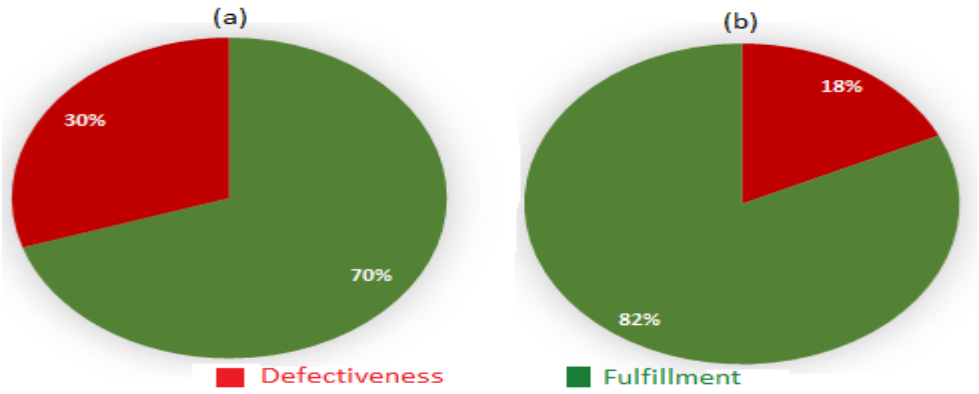

Figure 7.10. Overall architectural defects

7.2.3 NFR responsibilities evaluation (implementation-time)

This evaluation phase starts after the accomplishment of the implementation stage, in which the assigned NFR responsibilities are measured by the RSI metric. The evaluation effort in this phase is performed in three main steps as follows.

Step 1. *Constructing RST*

To calculate RSI for both architectures (state 2), we need to construct for each quality attribute the corresponding RST (see section 6.2.2.1). Figure 7.11 illustrates the RST of the performance attribute of SA1. The main responsibility called *promoting performance* is divided into a set of sub-responsibilities (blue boxes), such as *improving bandwidth* and *enhancing databases response time*. The decomposition is performed iteratively until no further sub-responsibilities can be divided. Subsequently, we attached tactic nodes (yellow boxes) to each atomic sub-

responsibility (bottommost responsibilities) to construct the final RST tree as output. In the same way, we created RSTs for the other quality attributes.

Step 2. *Calculating RSI*
Calculating RSI consists of performing a set of sub-steps sequentially starting from the RST's bottommost nodes until we reach the top responsibility (see section 6.2.2.2). Figure 7.11 shows an example of calculating the performance's RSI of the web auction application as follows.

Step 2.1. A judgment score (-1, -2, 1, or 2) is attributed to each tactic node according to the satisfaction degree of the upper sub-responsibilities. This step requires the intervention of architects and evaluators to attribute these scores (in this experiment, this task is performed by us). This can be achieved, for instance, by analyzing and inspecting the source code, reverse engineering (generating UML class diagrams from code), and testing to verify the implementation and the achievement of tactics. For example, the *improve bandwidth* sub-responsibility is judged by checking the size of web pages, their loading time, and the implementation of the *reducing HTTP requests* tactic. The *enhance database response* sub-responsibility is judged based on performing a source code analysis to verify the implementation of the *Batching queries* and *connection pooling* tactics.

Step 2.2. The weight of each tactic is calculated using eq. (10). For instance, the caching tactic consists of two NFR rules:

- R1: enabling cache-control for images, scripts, and styles. This rule belongs to MG1 (weight=0.527).
- R2: using CSS sprites to reduce round trips between servers and clients. This rule belongs to MG2 (weight= 0.3).

Therefore, the weight of caching tactic= 0.527+0.3≈ 0.83. It should be noted that this step and all the succeeding steps are performed automatically.

Step 2.3. The attribute values are calculated for each tactic node using eq. (11). For example, *the connection pooling* tactic= 2 * 0.95= 1.9.

Step 2.4. The attribute values of atomic sub-responsibilities are propagated from tactic nodes with regard to the contribution weights using eq. (12) and eq. (13). For example, the *enhance DB response* sub-responsibility= (1 *0.63 * 0.25) + (2 * 0.95 * 0.75) = 1.58.

Step 2.5. The attribute values are propagated from sub-responsibilities to parent sub-responsibilities (e.g., *improving bandwidth*) according to the decomposition weights using eq. (14) and eq. (15). For example, the propagation of attribute values to the upper sub-responsibility called Improve bandwidth= (0.83 * 0.27) + (0.17 * 0.05) + (1.76 * 0.56) + (0.37 * 0.12) = 1.26.

Step 2.6. The RSI of the top responsibility is calculated by the same propagation technique using eq. (14) and eq. (16), in which we found RSI=**0.78.**

Moreover, we found that RSI=0.73 if all tactics are moderately satisfied and 1.46 if all tactics are strongly satisfied. The obtained results mean that NFR responsibilities assigned to performance are slightly better than moderate satisfaction, which means that performance is promoted to a reasonable extent. However, RSI is considerably lesser than the strongly satisfied value because some tactics are moderately satisfied, and others are evidently unsatisfied, such as *multithreading* and *choosing the most performant server* tactics.

Similarly, we calculated RSI for other quality attributes (Table 7.8). It is obvious that the RSI of availability, maintainability (SA2), and extensibility are very close to the strong satisfaction values. However, the maintainability of SA1 is lesser than the optimum score because of some unsatisfied and moderately satisfied responsibilities.

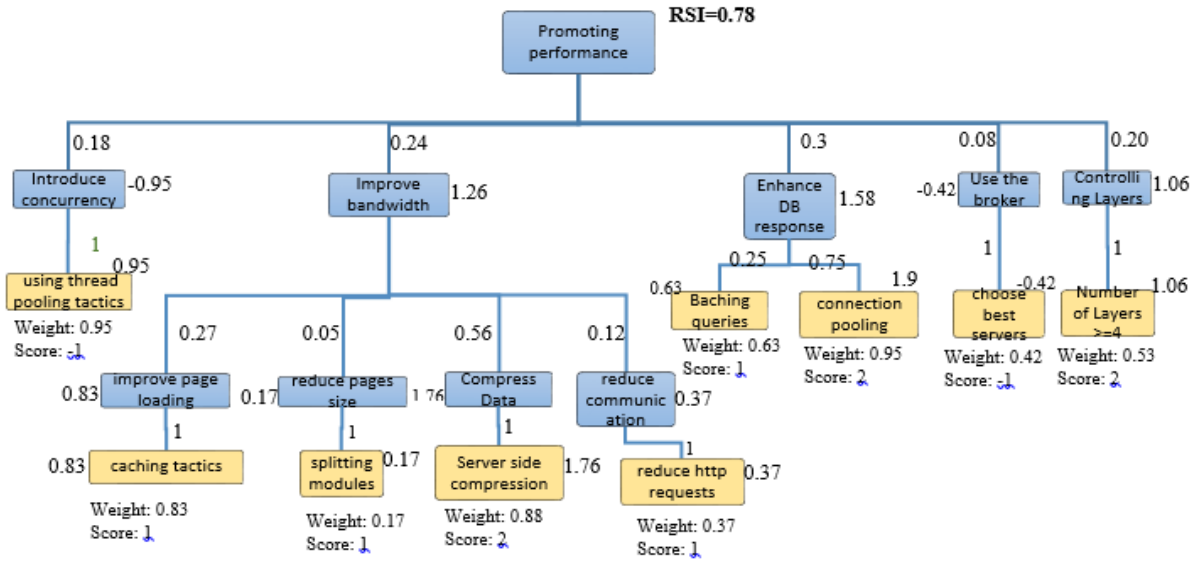

Figure 7.11. Calculating the RSI metric of the performance attribute

Quality attributes	Architecture	RSI	Moderate satisfaction	Strong satisfaction
Maintainability	SA1	1.6	1.07	2.14
Performance	SA1	0.78	0.73	1.46
Availability	SA1	1.7	0.88	1.75
Maintainability	SA2	1.55	0.8	1.6
Extensibility	SA2	1.54	0.79	1.58

Table 7.8. The obtained RSIs of SA1 and SA2.

Using the MS-QuAAF tool prototype to calculate RSI

The MS-QuAAF tool offers the user the possibility to calculate RSI automatically starting from step 2.2. Three simple steps are required to calculate RSI using this prototype tool as follows.

a) The Responsibility Satisfaction Tree (RST) of performance is converted into the corresponding XML representation (listing 7.3). Each responsibility and tactic node is mapped to the appropriate XML element. The *responsibility* XML element encompasses the attributes that depict the type of the responsibility, its name, and the parent node. The *tactic* XML element contains the attributes that depict the score and the weight assigned to the corresponding tactic node. Using the XML format to represent RST eased significantly the process of calculating RSI and reduced its complexity. It suffices to use an XML parser (we have used the Java DOM parser) to navigate through the different elements of the XML document and perform the required RSI computation.

b) The tool invites the user to load the appropriate XML document using the button *Load* (figure 7.12). By clicking this button, a highlighted XML text will be displayed in the dedicated text panel.

c) Once the XML document is loaded, the user can obtain the RSI of the desired quality attribute by clicking on the button *Calculate*. Additionally, the user can get the RSIs of the same quality attribute in case all tactics are moderately satisfied, unsatisfied, strongly satisfied, and strongly unsatisfied. All the results are displayed below the XML text panel.

```
<?xml version="1.0" encoding="UTF-8"?>
<qa name="Performance" arch_id="arch1">
<resp type="root" id="1" value="" name="promoting performance">
<resp type="sub_resp" id="2" linkto ="1" value="" name="introducing concurrency">
<tactic score="-1"  weight="0.95" value="" linkto="2">Using Thread pooling tactics </tactic>
</resp>
<resp type="sub_resp" id="3" linkto ="1" value="" name="improving bandwidth">
  <resp type="sub_resp" id="7" linkto ="3" value="" name="improving page loading">
  <tactic score="1"  weight="0.83"  value="" linkto="7">using thread pooling tactics </tactic>
...
  <resp type="sub_resp" id="10" linkto ="3" value="" name="reducing communications">
  <tactic score="1"  weight="0.37"  value="" linkto="10">reducing http requests </tactic>
  </resp>
</resp>
<resp type="sub_resp" id="4" linkto ="1" value="" name="enhancing DB response">
...
<tactic score="-1"  weight="0.42"  value="" linkto="5">choosing best server </tactic>
</resp>
<resp type="sub_resp" id="6" linkto ="1" value="" name="controlling layers">
<tactic score="2"  weight="0.53"  value="" linkto="6">number if layers is less or equal to 4 </tactic>
</resp>
</resp>
</qa>
```

Listing 7.3. An XML representation of the RST of performance

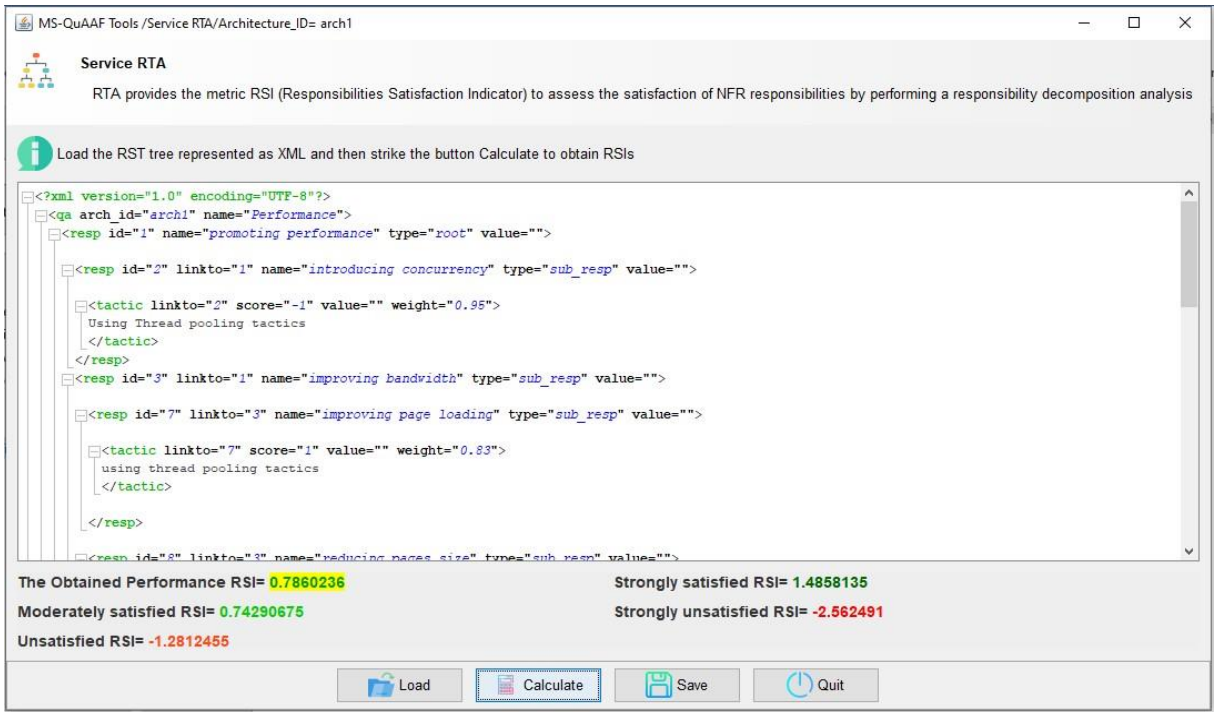

Figure 7.12. Calculating RSI of performance using the MS-QuAAF tool

7.2.4 Final assessment report

The last phase of the experiment consists of concluding the evaluation effort by producing the final assessment report (listing 7.4). For this purpose, the metrics ClosH and ClosV are used through the service LAFA. These metrics depend on the Euclidean distance to measure where the actual architecture quality stands. More specifically, we used these metrics in the experiment as follows.

Step1. *Creating the evaluation matrices*

For each project, we have created the evaluation matrices X1 (table 7.9) and X2 (table 7.10) as follows (see section 6.2.3.1)

Table 7.9. The evaluation matrix X1

Quality attributes	QuARF	RSI
Maintainability	0.67	1.6
Performance	0.63	0.78
Availability	0.79	1.7

Table 7.10. The evaluation matrix X2

Quality attributes	QuARF	RSI
Maintainability	0.77	1.55
Extensibility	0.87	1.54

X1= $(X1ij)_{3x2}$ and X2= $(X2ij)_{2x2}$ contain the assessment values obtained previously of the architectures SA1 and SA2, respectively, using QuARF and RSI. These matrices will be used by ClosH and ClosV in the next step to measure the closeness against the optimum values.

Step 2. *Calculating ClosH and ClosV*

Step 2.1. *Calculating ClosH*

The Euclidean distance between each horizontal vector of the evaluation matrices and the corresponding optimal vector is calculated using eq. (18). Table 7.11 shows the results of calculating ClosH of all quality attributes for the architectures SA1 and SA2. For instance, $ClosH_{performance}$ is calculated by measuring the closeness between the performance vector that contains the values <0.63, 0.78> and the optimal vector that contains the values <1, 1.46>. Similarly, we calculated the ClosH of all quality attributes.

Architecture	Quality attributes	QuARF	RSI	Optimum QuARF	Optimum RSI	ClosH
SA1	Maintainability	0.67	1.6	1	2.14	0.63
	Performance	0.63	0.78	1	1.46	0.77
	Availability	0.79	1.7	1	1.75	0.21
SA2	Maintainability	0.77	1.55	1	1.6	0.23
	Extensibility	0.87	1.54	1	1.58	0.13

Table 7.11. ClosH results

Step2.2. *Calculating ClosV*

Calculating ClosV consists of measuring the closeness between each vertical vector of the evaluation matrices and the corresponding optimal vector using eq. (18). Table 7.12 shows the results of calculating $ClosV_{QuARF}$ and $ClosV_{RSI}$ of the architectures SA1 and SA2. For example, $ClosV_{RSI}$ of SA1 is calculated by measuring the Euclidean distance between the vertical vector <1.6, 0.78, 1.7> of the matrix X1 and the optimal vector <2.14, 1.46, 1.75> (Table 7.11).

Architecture	$ClosV_{QuARF}$	$ClosV_{RSI}$
SA1	0.53	0.87
SA2	0.26	0.064

Table 7.12. ClosV results

Step 3. *Results interpretation and the final report*

ClosH permits to judge the satisfaction of each quality attribute inclusively at the design and implementation stages. Table 7.11 shows that the obtained ClosH of maintainability and performance of SA1 is averagely greater than zero, which means they are moderately close to the optimal values. This is mostly due to the obtained QuARF scores that indicate the existence of several architecture defects at design time, and thus the deviation from the specified architectural decisions. Besides, there are some NFR responsibilities that are not satisfied at the implementation stage, which contributed to increasing the distance from the optimal values. Figure 7.13 (b) illustrates the distance between the performance vector and the optimum vector ($ClosH_{performance}$=0.77). This distance is the largest in this experiment, which means that performance is the less satisfying quality attribute.

On the other hand, the obtained ClosH of availability, maintainability (SA2), and extensibility are sufficiently close to zero, which we consider as a very satisfactory result (especially for the extensibility). Mainly, this is due to the low defectiveness rate at design time and the satisfaction

of the most assigned NFR responsibilities of the implemented architecture. Figure 7.13 (a) illustrates a very short distance (ClosH $_{extensibility}$=0.13) between the extensibility vector and the optimum vector, which implies that extensibility is the most satisfying quality attribute in this experiment. Conclusively, ClosH converges to zero as QuARF and RSI increase and achieve the best scores.

ClosV permits to judge the overall architecture quality at the design stage and the implementation stages separately. Table 7.12 shows that the obtained ClosV$_{QuARF}$ of SA1 is somewhat greater than zero, which means that the overall designed architecture suffers from some architecture defects. These defects are mainly caused by the infringements of some important architectural decisions of the facets AF1 and AF2 related to the maintainability and performance attributes, respectively (Table 7.6). However, the attained ClosV$_{QuARF}$ of SA2 is close to zero to a certain extent, which means that the designed architecture is healthy despite a few architecture defects that affected the facets AF4 and AF5 related to the maintainability and extensibility attributes, respectively (Table 7.7).

On the other hand, the obtained ClosV$_{RSI}$ of SA1 is greater than zero, which means that the overall quality of the implemented architecture is somewhat far from the desired results. The defective architecture at the design time and therefore, the dissatisfaction of some important NFR responsibilities at the implementation stage (especially for maintainability and performance) have contributed to increasing the distance from the optimal values. Contrarily, the attained ClosV$_{RSI}$ of SA2 is very close to zero (ClosV$_{RSI}$=0.064), which means that the implemented architecture has achieved the optimal values. This is due to the very short Euclidean distance between the maintainability and extensibility vectors and the corresponding optimum vectors. Similar to ClosH, ClosV$_{QuARF}$ and ClosV$_{RSI}$ converge to zero as QuARF and RSI increase, respectively.

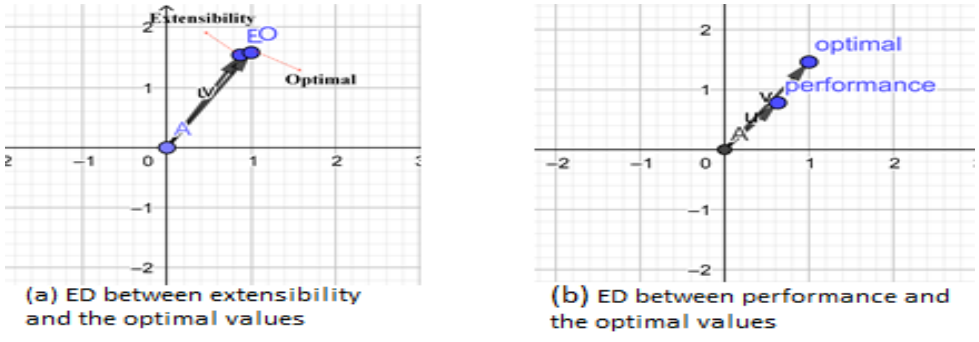

Figure 7.13. The ED between extensibility and performance vectors and the optimum vectors

The final report can be also generated as an XML document (listing 7.4). The elements encompassed within the tag *ClosH* describe the state of each target quality attribute. The *ED* (Euclidean Distance) attribute represents the value of ED of the parent quality attribute tag, whereas the *max_ED* attribute represents the worst-case scenario if all rules are completely violated (the optimal value is zero). The XML element QA_*Deviation* depicts the deviation

from the architectural decisions made to promote a quality attribute, whereas the element *QA_health* illustrates the healthiness of such attribute.

The XML element *ClosV* encompasses the elements that depict the overall architecture state at the design and implementation stages. These elements illustrate the severity of deviation of the designed architecture from the specified (the *design_deviation* XML element) one and the implemented architecture from the designed one (the *implementation_deviation* XML element).

```
<?xml version="1.0" encoding="UTF-8"?>
<arch id="arch1">
<evaluation_stage name="late_assessment">
<ClosH>
<QA name="performance" ED="0.77" Max_ED="4.14">
<comment>
The ED between performance and the optimum value is 0.77, wheras the max distance is 4.14</comment>
<QA_Deviation>
<severity> Moderatly close to the optimal value</severity>
<cause> Architectural defects at design time </cause>
<cause> responsibilities are not satisfied at the implementation stage</cause>
<details>
….
</details>
</QA_Deviation>
<qa_health> partially healthy</qa_health>
</QA>
<QA name="maintainability" ED="0.63" Max_ED="4.5">
.....
</ClosH>
<ClosV>
<quarf_overall value="0.53">
<comment> architecture suffers from some architecture defects </comment>
<design_deviation>
<severity> acceptable</severity>
<cause> Some rules of architecture facets AF1 and AF2 are violated</cause>
<details>
……
</details>
<design_health> relatively healthy</design_health>
....
<rsi_overall value="0.87">
<comment> Somewhat far from the desired results</comment>
<implementation_deviation>
<severity> acceptable</severity>
<cause> Some high weighted responsibilities are not satisfied</cause>
<details>
…..
</details>
<implementation_health> relatively healthy</implementation_health>
</implementation_deviation>
....
</arch>
```

Listing 7.4. The final evaluation report of SA1 structured as an XML document (Appendix C)

7.3 Answering the research questions

RQ1: Is the proposed framework capable of calculating the defectiveness of the designed architecture? As shown in section 7.2.2, the evaluation started after the accomplishment of the design stage, in which we extracted the concerned facets and subsequently regrouped design rules into weighted macro-groups. The experiment showed that the defined metrics of this stage are capable of calculating architecture defectiveness more accurately and equitably despite the specification language. More specifically, these metrics depend on the weight of rules to calculate defectiveness, which means that violating rules rated as extremely important do not have the same impact on the designed architecture as violating rules rated as moderately important. As a result, we found that architecture SA1 has a defectiveness rate (33%) greater than SA2 (18%) because SA1 has infringed some rules classified as extremely and strongly important contrarily to SA2 where most violated rules are ranked as moderately and slightly important.

RQ2: Is the proposed framework capable of estimating the satisfaction of NFR responsibilities of the implemented architecture? As shown in section 7.2.3, the evaluation started after the accomplishment of the implementation stage, in which we created an RST tree for each quality attribute to calculate the relevant RSI. The experiment demonstrated that the metric RSI is capable of estimating the satisfaction of the assigned NFR responsibilities more accurately and effectively by involving the decomposition, contribution, and tactic weights in a bottom-up measurement process. More precisely, tactics do not have the same weight, which implies their satisfaction and dissatisfactions do not have the same impact on the implemented architecture. Consequently, the obtained RSIs of SA1 and SA2 proved that dissatisfying highly ranked responsibilities have affected the architecture negatively more than dissatisfying low ranked responsibilities. These results strengthen the accuracy and effectiveness of using weighted RSTs in calculating RSI.

RQ3: Are the proposed metrics capable of deducing architecture deviations? As shown by section 7.2.4, the metrics ClosH and ClosV are capable of detecting two types of deviations. The former is the deviation of the designed architecture from the specified one. The latter is the deviation of the implemented architecture from the designed one. This type of deviation is called architecture erosion. Moreover, the deviation can be identified partially at the facet level using ClosH or entirely (the whole architecture) using ClosV. At the facet level, the deviation concerns only the mapped quality attribute, in which some architectural decisions taken to promote this attribute are violated. On the other hand, the overall deviation concerns the overall architecture with all targeted quality attribute at the design and implementation stages

RQ4: Does the proposed framework help in enhancing architecture quality? We believe that MS-QuAAF can contribute to improving the architecture quality on the condition that detected irregularities are adjusted to conform specifications (especially at the design stage where most of the changes are still inexpensive and possible). The metrics of the design stage permit

detecting violated rules, which can help to fix these rules before the implementation stage. The reparation procedure helps in boosting up the quality of the designed architecture, and hence the implemented architecture subsequently. Besides, the metric RSI can help in enhancing the implemented architecture by detecting unsatisfied NFR responsibilities. However, repairing flaws at this stage can be performed to a limited extent since carrying out major changes can be very expensive and hard to achieve.

7.4 Threats to validity

In this dissertation, we discuss three types of threats to validity: internal, external, and construct. To the internal validity, we can declare three main internal threats. First, the architecture evaluation performed through MS-QuAAF's services depends entirely on architecture specifications that constitute meta-models against which the conformance of the designed and implemented architecture is assessed. More specifically, the evaluation accuracy has a direct correlation with the quality and exactitude of these meta-models. Therefore, performing evaluation against inconsistent or wrongly defined meta-models will affect certainly the evaluation outcomes. To mitigate this threat, architecture specifications must be performed and reviewed by experienced architects to produce correct and consistent meta-models.

The second threat is the type of specification languages used to specify architecture. In this dissertation, the evaluation worked properly against OCL and Alloy meta-models since we have decent experience with these two languages (especially Alloy) that we find suitable for specifying rules and constraints. However, we expect difficulties with informal specifications due to the ambiguity of this type of specification (Jackson, 2019). To mitigate this threat, it would be better to specify architecture with formal languages that have the ability to express structures, relationships, properties, and constraints.

The third threat concerns assigning satisfaction scores to the RST's tactic nodes to evaluate the achievement of the NFR responsibilities (section 7.2.3). Attributing these scores is based on architects' estimation, which depends on their experience and the understanding level of the target architecture. Consequently, providing inaccurate scores will jeopardize the correctness of the subsequent evaluation steps, and thus obtaining erroneous RSIs.

To the external validity, the main external threat is the type and size of the selected architectures. In this dissertation, we have selected two different types of software architectures (web-based and desktop) with different sets of target quality attributes to be evaluated by MS-QuAAF. Although, these architectures are academic, small, and unsophisticated compared to commercial, large, and real-world software architectures, we believe that the proposed evaluation methodology and metrics can be generalized to be applied to any well-specified architecture despite its type, size, and complexity. At the same time, we cannot estimate the time required to evaluate large architectures with various design rules and numerous lines of code, neither the required evaluation effort.

To the construct validity, the main threat is excluding some AE factors (section 5.2.2), such as cost, schedule, and architects' performance and experience from the measures (e.g., the ability to calculate defectiveness and NFR responsibilities satisfaction) used to assess the effectiveness of MS-QuAAF. These factors may influence the evaluation process and its outcome. For instance, using the framework by experienced architects may give more accurate evaluation results than inexperienced ones. However, it is hard to quantify the experience and use it as a measure to judge the effectiveness of the framework. The cost (evaluation budget) factor is also excluded as a measure since the case study involves academic projects conducted in an academic non-industrial context. To minimize this threat (which affects the construct *effectiveness*) and ensuring that the experiment will provide a respectable degree of exactitude, we conducted an embedded case study that encompasses two units of analysis. Each unit represents a different type of software project that is specified with a different specification language, and designed and implemented by a different group using a different development technology. We evaluated each architecture separately using the same metrics, and we obtained the expected evaluation outcomes and inferences for both projects.

7.5 Conclusion

In this last chapter, we have provided an implementation of the framework MS-QuAAF through an embedded case study that encompasses two analysis units (architectures SA1 and SA2). The implementation consists of conducting an experimental evaluation that adheres to the protocol of the case study design indicated above. To gauge the effectiveness of the framework, we have used two different architectures specified with two different languages and developed using different technologies. The evaluation was performed after the accomplishment of the design and implementation stages using the proposed metric suite. Furthermore, we have developed a prototype tool to assist evaluators during the evaluation process. The latter is divided into four main phases. First, we have extracted architecture facets mapped to the target quality attributes. Second, we have assessed the designed architecture against the specified one. Third, we have assessed the implemented architecture through the goal decomposition analysis technique. Fourth, we have generated the final assessment report. The experimental evaluation had allowed us to validate the metrics suite proposed within the framework MS-QuAAF, and thus approved our contribution to the software architecture evaluation domain.

Chapter 8

Conclusion

This chapter concludes this dissertation by summarizing the key finding of this research by answering the research questions. Additionally, it discusses the contributions of the proposed approach, its limitation and weaknesses, and perspectives and future work.

Contents

8.1 Research aims and overall findings 166

8.2 Contributions 167

8.3 Limitations and perspectives 169

8.1 Research aims and overall findings

For many decades, software quality is considered very critical and central for organizations, and a key component for business success in a highly demanding and competitive software market. In this context, software architecture has been considered as the appropriate level to deal with quality attributes. As the demand for satisfying these attributes increases, the size and complexity of architecture increase. Researchers, as well as practitioners, have realized that the achievement of quality attributes is constrained by the software architecture of these systems and the architectural decisions made at this level. Various methods have been proposed to evaluate software architecture, especially at the early stages of the development process. However, according to the literature study that we had conducted, we found that these methods suffer from many drawbacks: a) the evaluation is performed, either at the design time or implementation stage, b) most methods support the evaluation of one or two quality attributes, especially quantitative methods, c) qualitative methods lack statistical significance, d) architecture erosion analysis is not included, and e) most methods lack tool support.

This research has attempted to overcome the stated shortcomings by proposing a new evaluation methodology within a multi-service evaluation framework called MS-QuAAF. Taking into account the research aims, we have been in confrontation with many research questions during the development of this framework. The answers to these questions reflect the main findings of this research. The first question states whether the framework is capable of calculating the defectiveness of architecture at the design stage. We have found that the defined metrics of this stage (QuARF, QuARD, OVF, and OVD.) are adept at calculating architecture defectiveness more accurately and equitably despite the specification language. This is due to the weights that we have introduced on design rules to calculate defects. In this connection, violating rules rated

as extremely important does not have the same impact on the designed architecture as violating rules rated as moderately important. The second question indicates whether the framework is able to estimate the satisfaction of NFR responsibilities after the accomplishment of the implementation stage. We have found that the metric RSI is capable of estimating the satisfaction of the assigned NFR responsibilities more accurately and effectively by involving the decomposition, contribution, and tactic weights in a bottom-up measurement process. The third question stipulates whether the framework is adept at deducing architecture deviations. We have found that the framework is capable of detecting two types of architecture deviations through the metrics ClosH and ClosV. The first type is the deviation of the designed architecture from the specified one. The second type is the deviation of the implemented architecture from the designed one. The fourth question indicates whether the framework is capable of enhancing the quality of software architecture. The framework can contribute to improving the architecture quality during the evaluation process on the condition that detected defects and irregularities are adjusted to conform to architecture specifications.

8.2 Contributions

This dissertation contributes to the component-based software architecture quality control and evaluation. Primarily, it addresses the problematic of software quality and architecture evaluation, in which two main issues were considered. The first issue concerns the increased size and complexity of the target architectures. The second issue concerns architecture evaluation and how to overwhelm the shortcomings stated during the literature study. Accordingly, we have proposed two modules within MS-QuAAF to resolve these issues. The facet projector module is dedicated to dealing with architecture complexity, whereas the quality evaluator module is dedicated to assessing architecture quality. The contributions of this dissertation can be classified as follows.

8.2.1 Architecture modeling and slicing contributions

Software architectures are so complex, which means that describing them in a one-dimensional fashion is impossible (Bass et al., 2012). This implies that software architectures are not flat but rather multi-dimensional entities that encompass multiple facets and views. The tremendous complexity can make the architecture evaluation even harder and trickier. Therefore, we should expand our view to handle the architecture from multiple sides and angles according to stakeholders' quality attributes. To overcome the complexity issues, we have proposed the concept of architecture facets. The essence of facets is to hide the information that is not necessary for the evaluation task to be undertaken; therefore, it exposes only elements of interest mapped to the quality attribute under assessment. More concretely, a facet encompasses only the architectural decisions made (topology, tactics, constraints, etc.) to promote this attribute.

To extract facets, we have proposed the concept of model projection depicted in detail in (Kadri et al., 2020, 2021a). The projection is a procedure that transforms a portion of the source meta-

models that specify the overall architecture into a new reduced destination meta-model that represents the facet. The transformation consists of extracting from the source model only elements of interest through executing a projection query. Model projection is a language-independent algorithm that allows reducing complexity and downsizing models iteratively. Therefore, we contribute to meta-model slicing and reduction techniques by proposing a projection procedure used to reduce the size and complexity of architectures despite their specification language. We believe that depicting software architecture as facets through projection can enhance understandability significantly, thus streamlining quality assessment and monitoring.

8.2.2 Architecture evaluation contributions

In this dissertation, we have proposed MS-QuAAF, a multi-service evaluation framework dedicated specifically to assessing software architectures quantitatively. In the context of architecture evaluation, we have made four primary contributions through the quality evaluator module as follows.

All-inclusiveness. MS-QuAAF is a generic framework that supports the evaluation of all quality attributes. The evaluation effort is independent of quality attributes and their relevant quality models. The independency is promoted by the proposed generic metrics that direct the ability of the framework to evaluate any inputted quality attribute. However, this is feasible as long as the architectural decisions supposed to promote these attributes are stated within the architecture specification. More specifically, architectural decisions constitute meta-models to which target architectures must adhere. Therefore, the evaluation consists of verifying the properties and the conformance of these architectures against the specified meta-models using the proposed metrics.

Continuous evaluation. Providing continuous architecture assessment is the evaluation philosophy of MS-QuAAF. Most of the proposed evaluation methods assess architecture at the early stages of the development process. Contrarily, MS-QuAAF has the ability to evaluate two architecture states that emerge respectively after the accomplishment of the design and implementation stages. This implies that the framework provides long evaluation support by covering two crucial development stages, and even after major maintenance activities. This is achievable because we defined for each architecture state the appropriate services and metrics.

NFR responsibilities assessment. In this dissertation, we have proposed a new concept called NFR responsibilities to evaluate implemented architectures. For each quality attribute, NFR responsibilities are decomposed iteratively to construct the relevant RST (Responsibilities Satisfaction Tree). The latter is used within a newly proposed technique called the responsibility decomposition analysis (inspired by the goal analysis technique) to assess architecture quantitatively after the accomplishment of the implementation stage. Introducing tactic,

contribution, and decomposition weights on RSTs have contributed significantly to increasing the accuracy of the obtained evaluation results.

Architecture deviation detection. MS-QuAAF incorporates a feature that allows detecting architecture erosion. In this dissertation, we contribute to the domain of architecture erosion by proposing a mathematical approach to detect architecture deviation. This approach relies on the Euclidean Distance to calculate the closeness between the obtained evaluation results and the optimum results. Two types of deviation can be estimated. The deviation of the designed architecture from the specified one, and the deviation of the implemented architecture from the designed one. Architects and evaluators can approve or disapprove deviations calculated by the proposed closeness measures. Accepted deviations can be addressed and treated in the next architecture releases.

8.2.3 Applicative contribution

A prototype tool that implements the architecture and philosophy of the framework has been developed to facilitate, expedite, and automatize the evaluation process. The tool is called *MS-QuAAF Tools* and it encompasses two main modules: the facet projector and the quality evaluator. The first module implements the concept of facet projection to assist architects in extracting facets from large meta-models. The second module implements the evaluation services RDA, RTA, and LAFA to calculate the proposed evaluation metrics, thus reducing the effort during the evaluation process.

8.3 Limitations and perspectives

Although the contributions that we have made through MS-QuAAF, some issues and limitations can be reported during evaluation.

The consistency of architecture specification perspective. The quality and exactitude of architecture specifications have a great impact on the accuracy of the obtained evaluation results. Therefore, performing evaluation against inconsistent or wrongly defined meta-models will affect certainly the evaluation outcomes. A language like Alloy allows checking the consistency of the specified meta-models through assertions, whereas most other languages do not incorporate this feature. Our perspective in this situation is to add support for verifying the consistency of the specified meta-models, at least for the most known specification languages. Adding this feature will strengthen the framework and makes its evaluation results more accurate and trustworthy.

The dependency on architecture specification perspective. The architecture evaluation performed through MS-QuAAF's services depends entirely on architecture specifications that constitute meta-models against which the conformance of the designed and implemented architecture is assessed. The All-inclusiveness feature means that the framework can assess any inputted quality attribute. However, this is achievable as long as the architectural decisions supposed to promote these attributes exist in the NFR catalog. Consequently, the framework is

unable to assess quality attributes that have no corresponding architecture specification in this catalog. Our perspective is to enable the assessment of these attributes by incorporating reverse engineering support that allows extracting the design from the source code. Subsequently, the extracted design will be assessed against the architectural tactics and patterns used usually to promote these attributes.

The responsibility satisfaction analysis perspective. In this dissertation, we have proposed the responsibility decomposition analysis to assess the implemented architecture by decomposing the top responsibility iteratively to construct the corresponding RST tree. Each RST is mapped to one quality attribute. However, it is difficult to apply this type of evaluation where top responsibilities are not easily mapped to a tree or graph structure. Moreover, assigning satisfaction scores to the RST's tactic nodes is based on architects' estimation, which depends on their experience and the understanding of the target architecture. Consequently, providing inaccurate scores will threaten the accuracy of the obtained evaluation results. For the first limitation, our perspective is to enrich the second service dedicated to assessing the implemented architecture with another evaluation approach that can be applied to responsibilities that cannot be mapped to a tree structure. For the second limitation, our perspective is to improve the responsibility decomposition analysis technique and define new approaches and tools (e.g., reverse engineering tools) to help architects in judging tactic nodes more accurately.

Applicative perspective. The MS-QuAAF tools prototype that we have developed provides the primary services to extract facets and calculate metrics. We would like to upgrade this prototype to a complete quality assessment and monitoring tool. First, the facet projector module in the current version offers the possibility to create facets semi-automatically. Our perspective is to make this creation a fully automatic one-step procedure. Second, the quality evaluator module allows only calculating metrics through the three evaluation services. Our perspective is to enrich the framework with other tools that can help architects during the architecture analysis, such as a reverse engineering tool that facilitates the comparison between the implemented and designed architecture. Third, converting the MS-QuAAF tools into an Eclipse plugin that can be integrated within other development environments, thus enhancing the quality of the final software product.

Architecture-Driven Modernization perspectives. In the context of Architecture-Driven Modernization (ADM), our perspective is to develop a new set of metrics to evaluate and monitor architecture quality during the modernization of large legacy systems. We have already integrated the concept of facet projection within ADM to facilitate quality control (Aouag et al., 2020). Future work will focus on developing this new set of metrics either as a part of MS-QuAAF or as a separate framework.

List of published papers

This section presents the list of papers that we have published during this dissertation.

1. International Conferences

Salim Kadri, Sofiane Aouag, Djalal Hedjazi - **Multi-level approach for controlling architecture quality with Alloy.** 2019 International Conference on Theoretical and Applicative Aspects of Computer Science (ICTAACS).

Sofiane Aouag, Salim Kadri, Djalal Hedjazi - **Towards architectural view-driven modernization.** 2020 International Conference on Advanced Aspects of Software Engineering (ICAASE).

2. International journals

Salim Kadri, Sofiane Aouag, Djalal Hedjazi - **Multi-view model-driven projection to facilitate the control of the evolution and quality of the architecture.** International Journal of Software Innovation (IJSI), 2020, vol. 8, no 4, p. 21-39.

Salim Kadri, Sofiane Aouag, Djalal Hedjazi - **An Incremental Model Projection Applied to Streamline Software Architecture Assessment and Monitoring.** International Journal of Information System Modeling and Design (IJISMD), 2021, vol. 12, no 3, p. 27-43.

Salim Kadri, Sofiane Aouag, Djalal Hedjazi - **MS-QuAAF: A generic evaluation framework for monitoring software architecture quality.** Information and Software Technology (IST), 2021, vol. 140, p. 106713.

Bibliography

About the Object Constraint Language Specification Version 2.4. (n.d.). Retrieved July 14, 2019, from https://www.omg.org/spec/OCL/

About the Unified Modeling Language Specification Version 1.3. (n.d.). Retrieved January 22, 2021, from https://www.omg.org/spec/UML/1.3

Abowd, G. D., Bass, L., Clements, P. C., Kazman, R., & Northrop, L. M. (1997). *Recommended Best Industrial Practice for Software Architecture Evaluation.* Undefined. https://www.semanticscholar.org/paper/Recommended-Best-Industrial-Practice-for-Software-Abowd-Bass/8cc49394dbe2a6cb49f85087583865020c7154bd

Albrecht, A. J., & Gaffney, J. E. (1983). Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, *6*, 639–648.

Allen, R., & Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *6*(3), 213–249.

Allen, R. J. (1997). *A Formal Approach to Software Architecture.* CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE.

Al-Qutaish, R. E. (2010). Quality models in software engineering literature: An analytical and comparative study. *Journal of American Science*, *6*(3), 166–175.

Aouag, S., Kadri, S., & Hedjazi, D. (2020). Towards architectural view-driven modernization. *2020 International Conference on Advanced Aspects of Software Engineering (ICAASE)*, 1–6.

Babar, M. A., Zhu, L., & Jeffery, R. (2004). A framework for classifying and comparing software architecture evaluation methods. *2004 Australian Software Engineering Conference. Proceedings.*, 309–318.

Bachmann, F., Bass, L., Klein, M., & Shelton, C. (2005). Designing software architectures to achieve quality attribute requirements. *IEE Proceedings - Software*, *152*(4), 153. https://doi.org/10.1049/ip-sen:20045037

Bansiya, J., & Davis, C. G. (2002). A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, *28*(1), 4–17. https://doi.org/10.1109/32.979986

Bass, L., Clement, P., & Kazman, R. (2012). *Software Architecture in Practice* (Third edition). Addison-Wesley.

Bass, L., Merson, P., & O'Brien, L. (2005). Quality attributes and service-oriented architectures. *Department of Defense, Technical Report September*.

Bengtsson, P., & Bosch, J. (1998). Scenario-based software architecture reengineering. *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*, 308–317. https://doi.org/10.1109/ICSR.1998.685756

Bengtsson, P., Lassing, N., Bosch, J., & van Vliet, H. (2004). Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, *69*(1), 129–147. https://doi.org/10.1016/S0164-1212(03)00080-3

Blouin, A., Moha, N., Baudry, B., Sahraoui, H., & Jézéquel, J.-M. (2015). Assessing the use of slicing-based visualizing techniques on the understanding of large metamodels. *Information and Software Technology*, *62*, 124–142. https://doi.org/10.1016/j.infsof.2015.02.007

Boehm, B. W., Brown, J. R., Energy, T. I. S. and, & Kaspar, H. (1978). *Characteristics of Software Quality*. North-Holland Publishing Company. https://books.google.dz/books?id=Cdm0AAAAIAAJ

Boehm, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative evaluation of software quality. *Proceedings of the 2nd International Conference on Software Engineering*, 592–605.

Boxer, P., & Kazman, R. (2017). Chapter 9—Analyzing the Architectures of Software-Intensive Ecosystems. In I. Mistrik, N. Ali, R. Kazman, J. Grundy, & B. Schmerl (Eds.), *Managing Trade-Offs in Adaptable Software Architectures* (pp. 203–222). Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-802855-1.00009-5

Bradac, M. G., Perry, D. E., & Votta, L. G. (1994). Prototyping a process monitoring experiment. *IEEE Transactions on Software Engineering*, *20*(10), 774–784.

Briand, L. C., Morasca, S., & Basili, V. R. (1996). Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, *22*(1), 68–86.

Carrozza, G., Pietrantuono, R., & Russo, S. (2018). A software quality framework for large-scale mission-critical systems engineering. *Information and Software Technology*, *102*, 100–116. https://doi.org/10.1016/j.infsof.2018.05.009

Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20*(6), 476–493. https://doi.org/10.1109/32.295895

Choi, M., Kim, I. J., Hong, J., & Kim, J. (2009). Component-based metrics applying the strength of dependency between classes. *Proceedings of the 2009 ACM Symposium on Applied Computing*, 530–536.

Chung, L., Hill, T., Legunsen, O., Sun, Z., Dsouza, A., & Supakkul, S. (2013). A goal-oriented simulation approach for obtaining good private cloud-based system architectures. *Journal of Systems and Software*, *86*(9), 2242–2262.

Chung, L., Nixon, B., Yu, E., & Mylopoulos, J. (2000). *Non-Functional Requirements in Software Engineering* (Vol. 5). https://doi.org/10.1007/978-1-4615-5269-7

Clemente, P. J., Hernández, J., Conejero, J. M., & Ortiz, G. (2011). Managing crosscutting concerns in component based systems using a model driven development approach. *Journal of Systems and Software*, *84*(6), 1032–1053. https://doi.org/10.1016/j.jss.2011.01.053

Clements, P. C. (2000). *Active reviews for intermediate designs*. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

Clements, P. C. (1996a). A survey of architecture description languages. *Proceedings of the 8th International Workshop on Software Specification and Design*, 16–25.

Clements, P. C. (1996b). A survey of architecture description languages. *Proceedings of the 8th International Workshop on Software Specification and Design*, 16–25.

Clements, P., Garlan, D., Little, R., Nord, R., & Stafford, J. (2003). Documenting software architectures: Views and beyond. *25th International Conference on Software Engineering, 2003. Proceedings.*, 740–741. https://doi.org/10.1109/ICSE.2003.1201264

Crnkovic, I., Chaudron, M., & Larsson, S. (2006). Component-Based Development Process and Component Lifecycle. *2006 International Conference on Software Engineering Advances (ICSEA'06)*, 44–44. https://doi.org/10.1109/ICSEA.2006.261300

Crnkovic, I., & Larsson, M. P. H. (2002). *Building reliable component-based software systems*. Artech House.

Crosby, P. B. (1980). *Quality is Free: The Art of Making Quality Certain*. New American Library. https://books.google.dz/books?id=abM4662_lF4C

Crosby, P. B. (1995). *Quality Without Tears: The Art of Hassle-Free Management*. McGraw-Hill Education. https://books.google.dz/books?id=lzv7kHpplKwC

Cunha, A., Garis, A., & Riesco, D. (2015). Translating Between Alloy Specifications and UML Class Diagrams Annotated with OCL. *Softw. Syst. Model.*, *14*(1), 5–25. https://doi.org/10.1007/s10270-013-0353-5

De Silva, L., & Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software*, *85*(1), 132–151.

de Silva, L., & Balasubramaniam, D. (2012). Controlling software architecture erosion: A survey. *Journal of Systems and Software*, *85*(1), 132–151. https://doi.org/10.1016/j.jss.2011.07.036

DeMarco, T. (1986). *Controlling software projects: Management, measurement, and estimates*. Prentice Hall PTR.

Deming, W. E. (1986). *Out of the Crisis*. Massachusetts Institute of Technology, Center for Advanced Engineering Study. https://books.google.dz/books?id=4qw8AAAAIAAJ

Dobrica, L., & Niemela, E. (2002). A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering*, *28*(7), 638–653. https://doi.org/10.1109/TSE.2002.1019479

Dori, D. (2011). Object-process methodology. In *Encyclopedia of Knowledge Management, Second Edition* (pp. 1208–1220). IGI Global.

Dromey, R. G. (1995). A model for software product quality. *IEEE Transactions on Software Engineering*, *21*(2), 146–162.

Falessi, D., Cantone, G., Kazman, R., & Kruchten, P. (2011). Decision-Making Techniques for Software Architecture Design: A Comparative Survey. *ACM Computing Surveys*, *43*, 33. https://doi.org/10.1145/1978802.1978812

Fenton, N., & Bieman, J. (2019). *Software metrics: A rigorous and practical approach*. CRC press.

France, R., Ray, I., Georg, G., & Ghosh, S. (2004). Aspect-oriented approach to early design modelling. *IEE Proceedings - Software*, *151*(4), 173–185. https://doi.org/10.1049/ip-sen:20040920

Gaedke, M., & Rehse, J. (2000). Supporting compositional reuse in component-based Web engineering. *Proceedings of the 2000 ACM Symposium on Applied Computing-Volume 2*, 927–933.

Galin, D. (2004). *Software Quality Assurance: From Theory to Implementation*. Pearson Education Limited. https://books.google.dz/books?id=p5jDETUc2K8C

Garlan, D., Monroe, R., & Wile, D. (2010). Acme: An architecture description interchange language. In *CASCON First Decade High Impact Papers* (pp. 159–173).

Garlan, D., & Perry, D. E. (1995). Introduction to the special issue on software architecture. *IEEE Trans. Software Eng.*, *21*(4), 269–274.

Grady, R. B. (1992). *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall. https://books.google.dz/books?id=6KNQAAAAMAAJ

Halstead, M. H. (1977). *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc.

Heineman, G., & Councill, W. (2001). *Component-Based Software Engineering: Putting the Pieces Together*.

IEEE Recommended Practice for Architectural Description for Software-Intensive Systems. (2000). *IEEE Std 1471-2000*, 1–30. https://doi.org/10.1109/IEEESTD.2000.91944

IEEE Standard for Software Quality Assurance Processes. (2014). *IEEE Std 730-2014 (Revision of IEEE Std 730-2002)*, 1–138. https://doi.org/10.1109/IEEESTD.2014.6835311

Ishikawa, K. (1985). *What is Total Quality Control? The Japanese Way*. Prentice-Hall. https://books.google.dz/books?id=ANaYjgEACAAJ

ISO 25010. (n.d.). Retrieved May 15, 2020, from https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

ISO/IEC/IEEE International Standard—Software, systems and enterprise – Architecture evaluation framework. (2019). *ISO/IEC/IEEE 42030:2019(E)*, 1–88. https://doi.org/10.1109/IEEESTD.2019.8767001

Jackson, D. (2012). *Software Abstractions: Logic, Language, and Analysis* (Revised edition). MIT press.

Jackson, D. (2019). Alloy: A language and tool for exploring software designs. *Communications of the ACM*, *62*, 66–76. https://doi.org/10.1145/3338843

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc.

Jacobson, I., Griss, M., & Jonsson, P. (1997). *Software Reuse: Architecture Process and Organization for Business Success*. ACM Press. https://books.google.dz/books?id=wrFQAAAAMAAJ

Johnson, W. L., & Harris, D. R. (1991). Sharing and reuse of requirements knowledge. *Proceedings of the 6th International Conference on Knowledge-Based Software Engineering*, 57–66.

Juran, J. M. (1964). *Managerial breakthrough: A new concept of the manager's job*. McGraw-Hill Companies.

Juran, J. M., & Godfrey, A. B. (1999). *Juran's Quality Handbook*. McGraw-Hill Education. https://books.google.dz/books?id=moURDQAAQBAJ

Kadri, S., Aouag, S., & Hedjazi, D. (2020). Multi-View Model-Driven Projection to Facilitate the Control of the Evolution and Quality of the Architecture. *International Journal of Software Innovation (IJSI)*, *8*(4), 21–39.

Kadri, S., Aouag, S., & Hedjazi, D. (2021a). An Incremental Model Projection Applied to Streamline Software Architecture Assessment and Monitoring. *International Journal of Information System Modeling and Design (IJISMD)*, *12*(3), 27–43.

Kadri, S., Aouag, S., & Hedjazi, D. (2021b). MS-QuAAF: A generic evaluation framework for monitoring software architecture quality. *Information and Software Technology*, *140*, 106713.

Kadri, S., Aouag, S., & Hedjazi, D. (2019). Multi-level approach for controlling architecture quality with Alloy. *2019 International Conference on Theoretical and Applicative Aspects of Computer Science (ICTAACS)*, *1*, 1–8.

Kaiya, H., Horai, H., & Saeki, M. (2002). AGORA: Attributed goal-oriented requirements analysis method. *Proceedings IEEE Joint International Conference on Requirements Engineering*, 13–22.

Kallel, S., Tibermacine, C., Kallel, S., Kacem, A. H., & Dony, C. (2018). Specification and automatic checking of architecture constraints on object oriented programs. *Information and Software Technology*, *101*, 16–31. https://doi.org/10.1016/j.infsof.2018.05.002

Kazman, R., Abowd, G., Bass, L., & Clements, P. (1996). Scenario-Based Analysis of Software Architecture. *Software, IEEE*, *13*, 47–55. https://doi.org/10.1109/52.542294

Kazman, R., Asundi, J., & Klein, M. (2001). Quantifying the costs and benefits of architectural decisions. *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, 297–306. https://doi.org/10.1109/ICSE.2001.919103

Kazman, R., Klein, M., & Clements, P. (2000). *ATAM: Method for architecture evaluation*. Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst.

Kelly, T., & Weaver, R. (2004a). The goal structuring notation–a safety argument notation. *Proc Dependable Syst Networks Workshop Assurance Cases*.

Kelly, T., & Weaver, R. (2004b). The goal structuring notation–a safety argument notation. *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 6.

Kitchenham, B., Pfleeger, S. L., & Fenton, N. (1995). Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, *21*(12), 929–944.

Kobayashi, N., Morisaki, S., Atsumi, N., & Yamamoto, S. (2016). Quantitative Non Functional Requirements Evaluation Using Softgoal Weight. *Journal of Internet Services and Information Security*, *6*, 37–46.

Kokune, A., Mizuno, M., Kadoya, K., & Yamamoto, S. (2007). FBCM: Strategy modeling method for the validation of software requirements. *Journal of Systems and Software*, *80*(3), 314–327. https://doi.org/10.1016/j.jss.2006.04.035

Koziolek, H. (2010). Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, *67*(8), 634–658. https://doi.org/10.1016/j.peva.2009.07.007

Kruchten, P. (2000). *The Rational Unified Process: An Introduction*. Addison-Wesley. https://books.google.dz/books?id=dJ5QAAAAMAAJ

Kruchten, P., Capilla, R., & Dueñas, J. (2009). The Decision View's Role in Software Architecture Practice. *IEEE Software*, *26*, 36–42. https://doi.org/10.1109/MS.2009.52

Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys (CSUR)*, *24*(2), 131–183.

Laddad, R. (2003). *AspectJ in Action: Practical Aspect-oriented Programming*. Manning. https://books.google.dz/books?id=2efVVV8G6oMC

Lorenz, M., & Kidd, J. (1994). *Object-oriented Software Metrics: A Practical Guide*. PTR Prentice Hall. https://books.google.dz/books?id=lsJnQgAACAAJ

Luckham, D. C., Kenney, J. J., Augustin, L. M., Vera, J., Bryan, D., & Mann, W. (1995). Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, *21*(4), 336–354.

Lung, C.-H., Bot, S., Kalaichelvan, K., & Kazman, R. (1997). An approach to software architecture analysis for evolution and reusability. *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, 15.

Magee, J., Dulay, N., Eisenbach, S., & Kramer, J. (1995). Specifying distributed software architectures. *European Software Engineering Conference*, 137–153.

Malhotra, R., & Chug, A. (2016). Software maintainability: Systematic literature review and current trends. *International Journal of Software Engineering and Knowledge Engineering*, *26*(08), 1221–1253.

Mary, S., & David, G. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

McCall, J. A., Richards, P. K., & Walters, G. F. (1977). *Factors in software quality. Volume i. Concepts and definitions of software quality*. GENERAL ELECTRIC CO SUNNYVALE CA.

Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, *26*(1), 70–93. https://doi.org/10.1109/32.825767

Meyer, B. (1992). Applying'design by contract'. *Computer*, *25*(10), 40–51.

Molter, G. (1999). Integrating SAAM in domain-centric and reuse-based development processes. *Proceedings of the 2nd Nordic Workshop on Software Architecture, Ronneby*, 1–10.

Mylopoulos, J., Chung, L., & Yu, E. (1999). From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, *42*(1), 31–37. https://doi.org/10.1145/291469.293165

O'Neill, B. (2014). *Elementary Differential Geometry*. Elsevier Science. https://books.google.dz/books?id=HrriBQAAQBAJ

O'Regan, G. (2014). *Introduction to Software Quality*. Springer Publishing Company, Incorporated.

Pérez, J., Ali, N., Carsí, J. A., & Ramos, I. (2006). Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In I. Gorton, G. T. Heineman, I. Crnković, H. W. Schmidt, J. A. Stafford, C. Szyperski, & K. Wallnau (Eds.), *Component-Based Software Engineering* (pp. 123–138). Springer. https://doi.org/10.1007/11783565_9

Perry, D. E., Sim, S. E., & Easterbrook, S. M. (2004). Case studies for software engineers. *Proceedings. 26th International Conference on Software Engineering*, 736–738.

Perry, D. E., & Wolf, A. L. (1992a). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, *17*(4), 40–52.

Perry, D. E., & Wolf, A. L. (1992b). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, *17*(4), 40–52. https://doi.org/10.1145/141874.141884

Pree, W. (1997). Component-based software development-a new paradigm in software engineering? *Proceedings of Joint 4th International Computer Science Conference and 4th Asia Pacific Software Engineering Conference*, 523–524.

Riaz, M., Mendes, E., & Tempero, E. (2009). *A systematic review of software maintainability prediction and metrics*. 367–377. https://doi.org/10.1109/ESEM.2009.5314233

Richters, M., & Gogolla, M. (2002). OCL: Syntax, semantics, and tools. In *Object Modeling with the OCL* (pp. 42–68). Springer.

Royce, W. W. (1987). Managing the development of large software systems: Concepts and techniques. *Proceedings of the 9th International Conference on Software Engineering*, 328–338.

Ruijters, E., & Stoelinga, M. (2015a). Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer Science Review*, *15*, 29–62.

Ruijters, E., & Stoelinga, M. (2015b). Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer Science Review*, *15–16*, 29–62. https://doi.org/10.1016/j.cosrev.2015.03.001

Rumbaugh, J., Jacobson, I., & Booch, G. (1999). The unified modeling language. *Reference Manual*.

Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131–164.

Runeson, P., Höst, M., Rainer, A., & Regnell, B. (2012). Case Study Research in Software Engineering—Guidelines and Examples. In *Case Study Research in Software Engineering: Guidelines and Examples*. https://doi.org/10.1002/9781118181034

Saaty, R. W. (1987). The analytic hierarchy process—What it is and how it is used. *Mathematical Modelling*, *9*(3), 161–176. https://doi.org/10.1016/0270-0255(87)90473-8

Saaty, T. L., & Vargas, L. G. (2012). *Models, Methods, Concepts & Applications of the Analytic Hierarchy Process* (2nd ed.). Springer US. https://doi.org/10.1007/978-1-4614-3597-6

Saito, S., & Yamamoto, S. (2006). The Incremental Goal Evolution Process Methodology. *BUSITAL*.

Shewhart, W. A. (1931). *Economic control of quality of manufactured product*. Macmillan And Co Ltd, London.

Shewhart, W. A. (1958). Nature and origin of standards of quality. *The Bell System Technical Journal*, *37*(1), 1–22.

Sobhy, D., Bahsoon, R., Minku, L., & Kazman, R. (2021). Evaluation of Software Architectures under Uncertainty: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology*.

Sommerville, I. (2011). *Software Engineering*. Pearson. https://books.google.dz/books?id=l0egcQAACAAJ

Spivey, J. M., & Abrial, J. (1992). *The Z notation*. Prentice Hall Hemel Hempstead.

Stein, D., Hanenberg, S., & Unland, R. (2002). *Designing Aspect-Oriented Crosscutting in UML*.

Subramanian, N., & Zalewski, J. (2014). Quantitative Assessment of Safety and Security of System Architectures for Cyberphysical Systems Using the NFR Approach. *IEEE Systems Journal*, *10*, 1–13. https://doi.org/10.1109/JSYST.2013.2294628

Svahnberg, M., Wohlin, C., Lundberg, L., & Mattsson, M. (2003). A quality-driven decision-support method for identifying software architecture candidates. *International Journal of Software Engineering and Knowledge Engineering*, *13*(05), 547–573. https://doi.org/10.1142/S0218194003001421

Szyperski, C., Gruntz, D., & Murer, S. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison Wesley Professional. https://books.google.dz/books?id=uNWGZwEACAAJ

Szyperski, C., & Pfister, C. (1996). Component-Oriented Programming: WCOP'96 Workshop Report. *Special Issues in Object-Oriented Programming: Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP'96, Linz*, 127–130.

Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead, E. J., Robbins, J. E., Nies, K. A., Oreizy, P., & Dubrow, D. L. (1996). A component-and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, *22*(6), 390–406.

Taylor, R. N., Medvidovic, N., & Dashofy, E. (2009). *Software Architecture: Foundations, Theory, and Practice*. Wiley. https://books.google.dz/books?id=npB5DwAAQBAJ

Tiwari, U. K., & Kumar, S. (2020). *Component-Based Software Engineering: Methods and Metrics*. CRC Press.

Welcome To UML Web Site! (n.d.). Retrieved April 8, 2020, from https://www.uml.org/

Westfall, L. (2016). *The certified software quality engineer handbook*. Quality Press.

Wohlin, C., & Aurum, A. (2015). Towards a decision-making structure for selecting a research design in empirical software engineering. *Empirical Software Engineering*, *20*(6), 1427–1455.

Yamamoto, S. (2015). An Approach for Evaluating Softgoals Using Weight. In I. Khalil, E. Neuhold, A. M. Tjoa, L. D. Xu, & I. You (Eds.), *Information and Communication Technology* (pp. 203–212). Springer International Publishing. https://doi.org/10.1007/978-3-319-24315-3_20

Zhou, Z., Zhi, Q., Morisaki, S., & Yamamoto, S. (2020). An Evaluation of Quantitative Non-Functional Requirements Assurance Using ArchiMate. *IEEE Access*, *8*, 72395–72410. https://doi.org/10.1109/ACCESS.2020.2987964

Appendix A
Alloy concise specification of the auction web architecture

```
1  module J2EE_Server4
2
3  abstract sig Component {coupledTo:Component}
4  sig intermediaryComponent extends Component {}
5  abstract sig layer {contains: some Component, message:one layer}
6  //sig intermediaryLayer extends layer {}
7  abstract sig Servlet_auction extends Component{calls:Ejb_auction}
8  abstract sig Ejb_auction extends Component{}
9  abstract sig Service_auction extends Component{}
10 one sig Server{divided_into:some layer, deliver:Response  ->  Broker,
addC: set intermediaryComponent}
11
12 sig article {registredIn:DB}
13 sig bidder{has: login}
14
15
16 ///////Declaring the broker/////////
17 sig Client {send:Request ->Broker}
18 sig Registry {contents: Service}
19 sig Service {resides:Registry}
20 abstract sig Message{}
21 sig Response extends Message {deliveredBy:Server}
22 sig Request extends Message {sentBy:Client}
23 one sig Broker {locates: some Service, ForwardResponseTo:some Client,
ForwardRequestTo:some Server}
24
25 ////////////Declaring Layers or Tiers/////////
26 //layer03
27 one sig Presentation_Layer extends layer {}
28 sig login extends Servlet_auction {}
29 sig Account {searches:search_article, buys:buy_article,
pays:pay_article}
30 sig search_article extends Servlet_auction {}
31 sig buy_article extends Servlet_auction{}
32 sig pay_article extends Servlet_auction {}
33 sig pay_DB extends Service_auction{update:article some-
>Connection_DB}
34 //Layer02
35 one sig Business_Layer extends layer {}
36 one sig login_Ejb extends Ejb_auction {connect:login_DB}
37 one sig search_Ejb extends  Ejb_auction  {searches:search_DB}
38 one sig buy_Ejb extends  Ejb_auction  {buys:buy_DB}
39 one sig pay_Ejb extends  Ejb_auction  {pays:pay_DB}
40
41 //Layer01
```

```
42 one sig Data_Layer extends layer {}
43 one sig DB extends Service_auction{}
44 sig login_DB extends Service_auction{openAccount:Account}
45 sig search_DB extends Service_auction{finds: article set -
>Connection_DB}
46 sig buy_DB extends Service_auction{update:article some-
>Connection_DB}
47
48
49 ///The presentation layer must contain only Servlet Components
50 fact {all s:Servlet_auction, e:Ejb_auction, r:Service_auction,
l:Presentation_Layer |s in l.contains && e not in l.contains  && r not
in l.contains}
51
52 ///The Buisiness layer must contain only EJBs Components
53 fact {all s:Servlet_auction, e:Ejb_auction, r:Service_auction,
l:Business_Layer|e in l.contains && s not in l.contains  && r not in
l.contains}
54 /*The Data access layer must contain only  Components that provide
55          DB connections, login, and execution of users' queries*/
56 fact {all s:Servlet_auction, e:Ejb_auction, r:Service_auction,
l:Data_Layer|r in l.contains && s not in l.contains  && e not in
l.contains}
57
58 // the architecture is divided into three layers, which are the
presentation, data, and Business Layers
59 fact {all l:layer, a: Server | l in a.divided_into}
60
61 // the number of layers should be less than or equal to 4
62 fact {one a: Server |   #a.divided_into<=5}
63
64 //Datasource,connection pooling to enhance performance
65 one sig Connection_DB{pooledTo:DataSource, execute:Query}//using
pooling to enhance performance
66 one sig DataSource{linkedTo:DB}//using data source
67 sig Query{belongTo:QueryType, executedAs: QueryExecution}
68 enum QueryType{Statement,PreparedStatement}
69 enum QueryExecution {Normal,Batch}
70 fact {all q:Query| q.belongTo=PreparedStatement}
71 fact {all q:Query| q.executedAs=Batch}
72
73 //////messages rules between layers
74 //the presentation layer can send messages to the business layer and
not vice versa
75 fact {all p:Presentation_Layer, b:Business_Layer| p.message in b}
76 //the business layer can send messages to the data layer and not vice
versa
77 fact {all  b:Business_Layer, d:Data_Layer| b.message in d}
78 //the data layer can send messages to the presentation layer and not
vice versa
79 fact {all  d:Data_Layer, p:Presentation_Layer| d.message in p}
80 //a layer cannot send a message to itself
81 fact {all p:Presentation_Layer,
b:Business_Layer,d:Data_Layer|p.message not in p &&
82 b.message not in b && d.message not in d}
83
```

```
84 /////Coupling between component
85 //coupling between 2 components should be lesser than 8
86 fact{all c:Component | #c.coupledTo <=7}
87 // a component cannot be coupled to it self
88 fact{all c:Component |c not in c.coupledTo}
89 //coupling degree between 2 component is greater than 7 imply
inserting an intermediary components
90 fact {all c:Component, c1:intermediaryComponent, s:Server
|#c.coupledTo<=7 implies c1 not in s.addC}
91
92
93 ///////////Some of the broker design rules
94 fact {all r:Registry, s:Service|s in r.contents && r in s.resides &&
contents=~resides}
95
96 //Rule 1: the number of requests must be equal to the number of
responses.
97 fact {all c:Client|all  s:Server |#c.send=#s.deliver && #
Response=#Request }
98
99 fact{one b:Broker|send.b in~sentBy &&#send.b =#~sentBy }
100
101//A server can't deliver a response without receiving a request
102fact {all s:Server |one b:Broker|some b.ForwardRequestTo implies
some s.deliver &&
103#(deliver.b.Response)=#(b.ForwardRequestTo) }
104//each client must receive a response if he/she sent a request
105 fact {all c:Client |one b:Broker|some b.ForwardResponseTo implies
some c.send &&
106 #(send.b.Request)=#(b.ForwardResponseTo) }
107
108//////////////////////
109// a bidder can buy only articles that can be found in the database
110 fact {all s:search_DB |all b:buy_DB | b.update in s.finds}
111
112 fact {all g:login |g.calls in login_Ejb}
113 fact {all s:buy_article |s.calls in buy_Ejb}
114 fact {all s:search_article |s.calls in search_Ejb}
115 fact {all s:pay_article |s.calls in pay_Ejb}
116
117// if a bidder pays some money, that means he bought some items
118 fact {all a:Account | some a.pays implies some a.buys &&
#a.buys=#a.pays}
119// if a bidder buy some items that's means he found those item in the
auction database
120 fact {all a:Account, s:search_DB| some a.buys implies some s.finds
}
121
122 pred test{}
123
124
125//Run predicate test
```

```
126 run test for 1 login, 1 search_article,  1 buy_article, 1
pay_article, 1 login_Ejb,1 search_Ejb, 1 buy_Ejb,
127 1 pay_Ejb, 1 Account, 1 login_DB, 1 search_DB,1 buy_DB, 1 pay_DB, 3
article, exactly 1 bidder, 128exactly 1 Client, 1 Server,1  Registry, 2
Message, 1 Service,1 Query,1 intermediaryComponent
```

Appendix B
OCL embedded specification of the visualization tool architecture

```
1 import ecore : 'http://www.eclipse.org/emf/2002/Ecore' ;
2
3 package testView : testView = 'http://www.example.org/testView'
4 {
5 class Architecture
6 {
7 attribute Arch_id : String[?];
8 attribute newAttribute : String[?];
9 property divided : Layer[+|1] { ordered };
10
11 /*
12 * the Recommended number of layer
13 */
14 invariant layer_count: self.divided->size()=3;
15
16 /*
17 * the maximum number of layer
18 */
19 invariant layer_max_count: self.divided->size()<=5;
20
21 /*
22 * All the layers are distinct
23 */
24 invariant distinct_layer: self.divided->forAll(l1,l2:Layer|l1<>l2 implies
l1.type<>l2.type);
25
26 /*
27 * the minimum layer capacity
28 */
29 invariant min_capacity: self.divided->forAll(l1:Layer|l1.capacity>0);
30
31 /*
32 * the maximum layer capacity
33 */
34 invariant max_capacity: self.divided->forAll(l1:Layer|l1.capacity<10);
35 }
36 class Layer
37 {
38 operation addComponent(added : Boolean[?]) : Boolean[?]
39 {
40 body: if getCapacity()<10 then added else not added endif;
41 }
42 operation removeComponent(removed : Boolean[?]) : Boolean[?]
43 {
44 body: if getCapacity()>1 then removed else not removed endif;
```

```
45 }
46 operation getCapacity() : ecore::EInt[?]
47 {
48 body: capacity;
49 }
50 operation sendMessage(m : Layer[?]) : Boolean[?]
51 {
52 body: Layer.allInstances()->forAll(p|if p.oclIsTypeOf(Reader_Layer) or
p.oclIsTypeOf(Converter_Layer) then sendMessage(p)
53 else
54 not sendMessage(p)
55 endif);
56 }
57 operation receiveMessage(m : Layer[?]) : Boolean[?]
58 {
59 body: Layer.allInstances()->forAll(p|if p.oclIsTypeOf(Converter_Layer) or
p.oclIsTypeOf(Renderer_Layer) then receiveMessage(p)
60 else
61 not receiveMessage(p)
62 endif);
63 }
64 attribute type : String[?];
65 attribute capacity : ecore::EInt[?];
66 }
67 class Renderer_Layer extends Layer
68 {
69 property contain#reside : Data_Renderer[+|1] { ordered };
70 property follow : Facade[1];
71
72 /*
73 * A Renderer Layer cannot send any message to the converter and reader layers
74 */
75 invariant
76 MsgDirection1:
77
78 Layer.allInstances()->forAll(p|if p.oclIsTypeOf(Reader_Layer) or
    p.oclIsTypeOf(Converter_Layer) then
79 not self.sendMessage(p)
80
81 else self.sendMessage(p) endif);
82
83 /*
84 * A Renderer Layer can receive messages only from the converter layer
85 */
86 invariant
87 MsgDirection2:
88
89 Layer.allInstances()->forAll(p|if p.oclIsTypeOf(Converter_Layer) then
90 self.receiveMessage(p)
91
92 else not self.receiveMessage(p) endif);
93
94 /*
95 * if the Renderer Layer receive a message from the converter layer, then it should
```

```
allow access to its components
96 */
97 invariant MsgReceived:
98 Layer.allInstances()->forAll(p|self.receiveMessage(p) implies
self.follow.allowAccess());
99
100 /*
101 * if the Renderer Layer allows access then it should select and provide the desired
Rendering component
102 */
103 invariant selectRenderer:
104 self.follow.allowAccess() implies self.follow.selectRenderer()->size()=1;
105
106 /*
107 * A Renderer Layer should contain only the Renderer objects, such as charts,
tables, etc.
108 */
109 invariant content:
110 self.contain-> forAll(p| p.oclIsTypeOf(Data_Renderer) );
111 }
112 class Converter_Layer extends Layer
113 {
114 property call : Renderer_Layer[1];
115 property contain#reside : Data_Converter[+|1] { ordered };
116
117 /*
118 * A Converter Layer can send messages only to the renderer layer
119 */
120 invariant
121 MsgDirection1:
122 let send : Renderer_Layer = self.call in
123 if self.call.oclIsTypeOf(Renderer_Layer)then sendMessage(send)else not
sendMessage(send) endif;
124
125 /*
126 * A Converter Layer cannot receive messages from the the Renderer layer
127 */
128 invariant MsgDirection2:
129 let send : Renderer_Layer = self.call in
130
131 not self.receiveMessage(send);
132
133 /*
134 * A Converter Layer should contain only the conversion objects
135 */
136 invariant content:
137 self.contain-> forAll(p| p.oclIsTypeOf(Data_Converter) );
138 }
139 class Reader_Layer extends Layer
140 {
141 property call : Converter_Layer[1];
142 property contain#reside : Data_Reader[+|1] { ordered };
143
```

```
144 /*
145 * A Reader Layer can send messages only to the converter layer
146 */
147 invariant
148 MsgDirection1:
149 let send : Converter_Layer = self.call in
150 if self.call.oclIsTypeOf(Converter_Layer)then sendMessage(send)else not
sendMessage(send) endif;
151
152 /*
153 * A Reader Layer cannot receive a message from the converter layer
154 */
155 invariant MsgDirection2:
156 let send : Converter_Layer = self.call in
157 not self.receiveMessage(send);
158
159 /*
160 * A Reader Layer should contain only the data readers
161 */
162 invariant content:
163 self.contain-> forAll(p| p.oclIsTypeOf(Data_Reader) );
164 }
165 class Data_Renderer
166 {
167 property reside#contain : Renderer_Layer[1];
168 attribute type : String[?];
169 property input#output : Data_Converter[1];
170
171 /*
172 * There is one Data converter attached to all renderer
173 */
174 invariant convertCount:
175 self.input->size()=1;
176
177 /*
178 * A Renderer object should reside within the Renderer_Layer
179 * (Reinforcing the Containing rules specified above)
180 */
181 invariant layer:
182 self.reside=Renderer_Layer;
183 }
184 class Data_Converter
185 {
186 operation convert(d : Data_Reader[?]) : Boolean[?];
187 property reside#contain : Converter_Layer[1];
188 property conform : DataAdapter[1];
189 property input#output : Data_Reader[1];
190 property output#input : Data_Renderer[1];
191
192 /*
193 * A data converted by the Converter object should be conform to the structure
194 * specified by the DataAdapter Interface
```

```
195 */
196 invariant
197 conformTo:
198 if self.convert(self.input)=true then self.conform.isConform()
199 else
200 not self.conform.isConform() endif;
201
202 /*
203 * A converter Component can handle one data reader at time
204 */
205 invariant ReaderCount:
206 self.input->size()=1;
207
208 /*
209 * A data converter output can be rendered by one renderer at time
210 */
211 invariant outputCount:
212 self.output->size()=1;
213
214 /*
215 * A data converter object should reside within the Converter_Layer
216 * (Reinforcing the Containing rules specified above)
217 */
218 invariant layer:
219 self.reside=Converter_Layer;
220 }
221 class Data_Reader
222 {
223 property reside#contain : Reader_Layer[1];
224 property output#input : Data_Converter[1];
225
226 /*
227 * All data reader objects are treated by one data converter
228 *
229 */
230 invariant outputCount:
231 self.output->size()=1;
232
233 /*
234 * A Reader object r should reside within the Reader_Layer
235 * (Reinforcing the Containing rules specified above)
236 */
237 invariant layer:
238 self.reside=Reader_Layer;
239 }
240 abstract class DataAdapter { interface }
241 {
242 /* produced data by the data converter component should be conform to the common
tabular format */
243 operation isConform() : Boolean[?]
244 {
245 body: if isTabular=true then isConform() else not isConform() endif;
```

```
246 }
247 operation getColumnCount() : ecore::EInt[?]
248 {
249 body: ColumnCount;
250 }
251 operation getRowCount() : ecore::EInt[?]
252 {
253 body: RowCount;
254 }
255 attribute ColumnCount : ecore::EInt[?];
256 attribute RowCount : ecore::EInt[?];
257 attribute isTabular : Boolean[?];
258
259 /*
260 * The number of columns of the tabular format should be greater than zero
261 */
262 invariant max_column_count:
263 ColumnCount>0;
264
265 /*
266 * the number of rows producer by the data converter should not be equal to zero
267 */
268 invariant max_row_count:
269 RowCount>0;
270
271 /*
272 * data should be always converted to a tabular format
273 */
274 invariant tabular:
```

Appendix C
The general XML structure of the final evaluation report

```
<?xml version="1.0" encoding="UTF-8"?>
<arch id="arch1">
<evaluation_stage name="late_assessment">
<ClosH>
<QA name="performance" ED="0.77" Max_ED="4.14">
<comment>
The ED between performance and the optimum value is 0.77, wheras the max
distance is 4.14
</comment>
<QA_Deviation>
<severity>
Moderatly close to the optimal value
</severity>
<cause>
Architectural defects at design time
</cause>
<cause>
responsibilities are not satisfied at the implementation stage
</cause>
<details>
...
</details>
</QA_Deviation>
<qa_health> partially healthy</qa_health>
</QA>
<QA name="maintainability" ED="0.63" Max_ED="4.5">
<comment>
The ED between performance and the optimum value is 0.77, wheras the max
distance is 4.14
</comment>
<QA_Deviation>
<severity>
Moderatly close to the optimal value
</severity>
<cause>
Architectural defects at design time and some NFR responsibilities
are not satisfied at the implementation stage...
</cause>
</QA_Deviation>
</QA>
</ClosH>
<ClosV>
<quarf_overall value="0.53">
<comment>
```

```
architecture suffers from some architecture defects
</comment>
<design_deviation>
<severity>
acceptable
</severity>
<cause>
Some rules of architecture facets AF1 and AF2 are violated
</cause>
<details>
...
</details>
<design_health> relatively healthy</design_health>
</design_deviation>
</quarf_overall >
<rsi_overall value="0.87">
<comment>
Somewhat far from the desired results
</comment>
<implementation_deviation>
<severity>
acceptable
</severity>
<cause>
Some high weighted responsibilities are not satisfied
</cause>
<details>
...
</details>
<implementation_health> relatively healthy</implementation_health>
</implementation_deviation>
</rsi_overall >
</rsi_overall>
</evaluation_stage>
</arch>
```