

FOUNDATIONS OF SYSTEMS AND PROPERTIES: METHODOLOGICAL SUPPORT FOR MODELING PROPERTIES OF SOFTWARE-INTENSIVE SYSTEMS

THÈSE N° 3013 (2004)

PRÉSENTÉE À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Institut d'informatique fondamentale

SECTION DES SYSTÈMES DE COMMUNICATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Otto PREISS

M.Sc. in Computer Science, University of Colorado, Etats-Unis
et de nationalité autrichienne

acceptée sur proposition du jury:

Prof. A. Wegmann, directeur de thèse
Prof. I. Crnkovic, rapporteur
Dr P. Kolb, rapporteur
Prof. S. Spaccapietra, rapporteur

Lausanne, EPFL
2004

Abstract

Engineering of software-intensive systems is concerned with the creation and evolution of systems that shall exhibit desired properties in their execution as well as development environment. In this context, the motivation of this thesis, derived from current development practice, was twofold. Firstly, software development methods are increasingly required to extend their scope of applicability towards systems engineering. As a consequence, their modeling approaches must be able to cope with a larger diversity of systems and consequently a larger diversity of properties. But these approaches still need to provide a smooth transition to software modeling. Secondly, non-functional properties, which are largely a result of this implicit systems scope, play a major role in the way we design our software-intensive systems. The conceptual aids of current development methods, however, are still less mature in their explicit support for non-functional properties compared with their ability to support functional ones.

The principal objective of this thesis is to contribute toward an improved model-based treatment of non-functional properties in development methods. Because we cannot discuss properties independently of the objects they are ascribed to, this objective amounts to a progression from modeling of software and its properties to modeling of interrelated systems and their properties.

To address this aim a philosophy of properties and systems is proposed. The philosophy is expressed as a holistic conceptual model of properties and/of systems. It is complemented with some basic rules, which we call tenets. Tenets formulate how we use the philosophical knowledge. The conceptual model offers the foundations for a more generalized understanding of those fundamentally different types of systems and different types of properties that are relevant in software-intensive systems engineering. The generality of our holistic model draws the benefits from our investigations in the areas of systems science, cognitive science, and basic philosophy. The model helps to scrutinize and make sense of the large amount of data in the literature about “non-functional” issues in software engineering. The model is applicable in the derivation of methodological building blocks that can be incorporated into development methods. The building blocks include (a) a general model to discover stakeholders and properties for a given system, (b) a principled manner to trace the fundamentally different types of properties through hierarchies of systems, and (c) a proposal for the representation of systems, their properties and property traces in the UML. The concrete application of the gained knowledge to software engineering results in a proposal for a context-sensitive, customizable quality attribute model. It also results in a proposal on how to structure quality descriptions of software components. In order for such descriptions to be standardized and possibly tool-automated, this thesis proposes to utilize the Reusable Asset Specification and suggests alternatives for its XML-based representation.

Version Abrégée

L'ingénierie de systèmes informatisés concerne la création et l'évolution de systèmes impliquant de nombreux logiciels. Ces systèmes exhibent des propriétés à la fois dans leur environnement d'exécution et de développement. Dans ce contexte, la motivation de cette thèse, basée sur des méthodes de développement actuelles, est double. Premièrement, il y a une demande grandissante pour que les méthodes de génie logiciel soient étendues et applicables à l'ingénierie de systèmes. Pour satisfaire cette demande, les méthodes de modélisation issues du génie logiciel doivent supporter une plus grande diversité de systèmes, et par conséquent une diversité de propriétés plus grande, tout en préservant une transition aisée vers la modélisation de logiciel. Deuxièmement, les propriétés non fonctionnelles, qui résultent largement de ce cadre implicite des systèmes, jouent un rôle majeur dans la manière de concevoir les systèmes impliquant de nombreux logiciels. Cependant, les méthodes de développement actuelles n'offrent qu'une aide conceptuelle limitée pour les propriétés non fonctionnelles et bien moins mature que pour des propriétés fonctionnelles.

L'objectif principal de cette thèse est de contribuer à l'amélioration du traitement des propriétés non fonctionnelles dans les méthodes de développement, en proposant une méthode systématique et basée sur des modèles. A cause de la dépendance mutuelle entre système et propriétés, cet objectif se traduit par une progression de la compréhension et de la modélisation du logiciel et de ses propriétés jusqu'à la compréhension et la modélisation de systèmes interdépendants et de leurs propriétés.

Une philosophie de propriétés et de systèmes est proposée dans ce but. Cette philosophie s'exprime en termes d'un modèle conceptuel et holistique de propriétés et/de systèmes et est complétée par des principes de base qui forment nos états d'esprit. Ce modèle conceptuel donne les fondations pour une compréhension plus généralisée des types fondamentalement différents de systèmes ainsi que des différents types de propriétés, tous deux étant pertinents dans l'ingénierie de systèmes impliquant de nombreux logiciels. Ce modèle holistique est général, grâce à nos recherches dans les domaines de la science de systèmes, de la science cognitive et de la philosophie de base. Sa valeur se montre dans le support à scruter et à synthétiser une grande quantité de données trouvées dans la littérature traitant de problèmes «non fonctionnels» en génie logiciel. Son applicabilité se montre dans la définition de blocs méthodologiques fondamentaux qui peuvent être incorporés dans les méthodes de développement. Ces blocs fondamentaux incluent un modèle général qui permet de découvrir les porteurs de tutelle et les propriétés d'un système donné, une manière explicite de retrouver les propriétés fondamentalement différentes à travers des hiérarchies de systèmes, ainsi qu'une proposition pour la représentation de systèmes, de leurs propriétés et des traces de propriété en UML. L'application concrète des connaissances acquises au génie logiciel résulte en une proposition d'un modèle de qualité adaptable aux besoins de clients, dépendent du contexte, ainsi qu'en une

proposition sur la manière de structurer des descriptions de qualité de composants logiciels. Pour que ces descriptions soient standardisées voire automatisées dans des outils de développement, cette thèse propose d'utiliser la spécification RAS (*angl.* Reusable Asset Specification) et suggère des alternatives pour sa représentation basée sur XML.

Acknowledgements

As with any such acknowledgement I have no illusion that this one will be complete. I can only hope that whoever would have expected to find her or his name here, but does not, shows leniency to me.

I want to express my deep and honest gratitude to Prof. Alain Wegmann. He has made it possible for me to dive into material and scientific strands, of which I did not even know they existed. He challenged many basics, which we as engineers take for granted, and he gently directed me to develop a deeper yet more general understanding of them. This certainly broadened my intellectual horizon greatly and it was extremely exciting, although, I must admit, I felt I would not progress at times, because I developed a habit of leaving no stone unturned. Prof. Alain Wegmann provided guidance when I wanted to be guided and most of all he was truly supportive, open, and very enthusiastic over all the years. Thank you, Alain!

Next, I am grateful to my jury members Prof. Ivica Crnkovic, Prof. Stefano Spaccapietra, and Dr. Peter Kolb for having accepted to serve on my PhD committee. I realize that this acceptance goes along with a considerable time investment to read and evaluate my work. It seemed to be purposeful coincidence that Prof. Ivica Crnkovic and myself met in academic settings such as conferences and workshops in the context of component-based software engineering and only later found out that we both have an ABB history. I am indebted to both Prof. Ivica Crnkovic and Dr. Peter Kolb for their timely motivation injections to “keep on going” despite an environment in which it is not easy to pursue a PhD.

I am especially grateful to my employer ABB. In fact, it is not the anonymous employer ABB, but the superiors and co-workers who have provided me with an environment that allowed me to pursue a PhD. In particular, my immediate superiors over the last few years Dr. Paolo Conti, Markus Greiner, Dr. Stefan Ramseier, Alban Frei, Dr. Kai Mossig, and Dr. Bernhard Eschermann have all been very supportive and even pushed me to be egoistic enough to put my PhD work on high priority. I want to thank Dr. Peter Kolb, who in his position as a global R&D program manager, enabled funding of my activities, at times when funding was really not easy to be obtained. I am grateful also to my fellow researchers at the ABB research center in Dättwil. Since most of them have gone through the emotional rollercoaster of writing a PhD thesis, they knew well when to push and when to pause. In this context, I would like to express my special thanks to three colleagues Dr. Tatjana Kostic, Dr. Johann Obermaier, and Dr. Christian Frei.

I would like to say “thank you!” to Angela Devenoge and Jean-Pierre Dupertuis in their respective role of secretary and IT system manager for the EPFL laboratories abbreviated “lams” and “ica”. Especially as an external PhD student, one learns to appreciate the invaluable help of responsive and competent people when it comes to finding ones way through administrative and IT jungles.

I want to acknowledge in a collective way many researchers who motivated me to look into “non-functionality” and, in writing or in discussions, confirmed a prevalent conceptual confusion in that area, a confusion that justified work towards a more holistic understanding and conceptual cleansing. Along the same lines of thinking, I am thankful to the members of the “lams”, with whom I had too few, but interesting discussions on philosophical or systems science related topics. I am also grateful for their critical assessment of my dry run of the PhD presentation.

Most of all, I am deeply grateful to my family. My daughters Laura and Jessica were incredibly patient with a daddy who seemed to be married to an office desk (as my wife Renata once said). But they were and are my greatest source of energy, a source I would never like to miss. “And yes, Renata, I will now give up my exclusive right to using our office room at home”. Without my wife, and this she knows, I could never have come that far and this not only refers to writing a PhD thesis.

Table of Contents

Abstract	i
Version Abrégée	ii
Acknowledgements	v
Table of Contents	vii
List of Figures	x
List of Tables	xiii

1 Introduction	1
1.1 Motivation.....	1
1.1.1 Support for Systems Development.....	2
1.1.2 Support For Non-Functional Properties.....	3
1.1.3 Customizability of Development Methods.....	4
1.2 Research Problem.....	5
1.3 Solution Approach.....	6
1.4 Research Method.....	10
1.5 Contribution.....	11
1.6 Roadmap.....	13

Part I

Context	15
----------------------	-----------

2 Systems- and Software Engineering	16
2.1 The Discipline of Systems Engineering.....	16
2.2 The Discipline of Software Engineering.....	18
2.2.1 Software Architecture.....	19
2.2.2 Component-Based Software Engineering.....	20
2.3 UML.....	22
2.3.1 UML Overview.....	22
2.3.2 UML Extension Mechanisms.....	25
2.4 Development Methods.....	26
2.4.1 Systems Development Methods.....	27
2.4.2 Software Development Methods.....	31
2.5 An Envisioned Systems Engineering Modeling Standard.....	35

3	The Different Flavors of Properties of Systems	38
3.1	The Systems Engineering Viewpoint	40
3.1.1	Properties and Classifications of Requirements in Systems Engineering	40
3.2	The Software Engineering Viewpoint	43
3.2.1	Quality Models and Requirements Classifications in Software Engineering.....	46
	Summary of Part I.....	51
Part II		
Methodological Building Blocks In Dealing with Hierarchical Systems and their Properties		
		53
4	Philosophy of Properties and Systems	54
4.1	Modeling.....	55
4.2	Properties	59
4.2.1	The Philosophical View On Properties	59
4.2.2	The Cognitive View On Properties	62
4.2.3	Properties - Conclusions.....	66
4.3	Systems.....	68
4.3.1	Concrete and Conceptual Systems	68
4.3.2	A General Model for Concrete Systems.....	79
4.3.3	Concrete versus Conceptual Decompositions	81
4.3.4	Systems - Conclusions.....	84
4.4	A Model of Systems <i>and</i> Properties	85
5	Methodological Building Blocks Derived from Fundamentals.....	87
5.1	The 2-2-2 Model and its Use	87
5.1.1	The 2-2-2 Model.....	87
5.1.2	Stakeholder Discovery and Classification.....	90
5.2	Decomposing and Tracing Properties	93
5.2.1	Realization-Oriented Decomposition Versus Other Forms of Property Decompositions	94
5.2.2	Realization-Oriented Traceability Patterns	95
5.2.3	Mutual Property Influences	103
5.3	UML as a Systems Modeling Language.....	103
5.3.1	Organization of a System Description.....	104
5.3.2	UML Extension Proposal	105
	Summary of Part II.....	119

Part III

Application to Software Engineering	121
6 Quality Attributes in Software Engineering Revisited.....	124
6.1 Proposed Core Terminology in Context with Non-Functional Properties.....	124
6.2 A Flexible Quality Model.....	126
6.2.1 Possibility Matrix for Quality Attributes.....	130
6.3 Standards and Related Research Revisited.....	132
6.3.1 ISO 9126 Revisited.....	132
6.3.2 Related Work.....	133
7 Quality Description of Software Components.....	134
7.1 Data Sheet for Software Components.....	134
7.1.1 Software Component Contracts.....	135
7.1.2 Conceptual Model of Quality Description.....	137
7.1.3 Determining Quality Attributes.....	139
7.2 Extending the RAS Metamodel with Quality Descriptions.....	145
7.2.1 The RAS Logical Model.....	147
7.2.2 Quality Description Extensions to the RAS.....	149
7.2.3 Proposal for a RAS Physical Model.....	152
7.3 ABB RAS Profiles.....	153
7.3.1 Enforcing AIP-Related Asset Descriptions.....	156
Summary of Part III	158
8 Conclusion	159
8.1 Summary of Results.....	159
8.2 Future Work.....	160
8.2.1 UML Extensions Necessitating More Than Lightweight Profiles.....	160
8.2.2 Modeling Metrics.....	162
8.2.3 Gaining Experience with Quality Attribute Descriptions of Reusable Assets ...	162
8.2.4 Enhancement of Development Methods and Tool Support.....	163
Bibliographic References.....	165
Appendix A: Glossary of Terms and Abbreviations.....	172
Appendix B: XML Schema Alternatives for the RAS.....	179
Appendix C: Complete Meta-Model of System and Properties.....	187

List of Figures

Figure 1-1: A (Partial) General Model for Systems	7
Figure 1-2: Pieces of Solution Approach	9
Figure 1-3: Systems Inquiry as Proposed by Banathy [10]	10
Figure 1-4: Chapter Layout and Influence Relationships	13
Figure 2-1: Scope of Systems- vs. Software Engineering	18
Figure 2-2: UML Meta-Modeling Architecture	23
Figure 2-3: Packages of the UML Semantics Specification	24
Figure 2-4: An OPD and Equivalent OPL Text of an Elevator Checkup Case Study	30
Figure 2-5: Workflow and Model Artifacts of UML Components	33
Figure 2-6: Kobra Specification and Realization Models (Source [5])	34
Figure 3-1: Positioning of Standards	39
Figure 3-2: Requirement Types as Defined by the ANSI/EIA 632 (Adapted from [94])	41
Figure 3-3: Sommerville’s Types of Non-Functional Requirement (Source [128])	43
Figure 3-4: Basic Rational behind the ISO 9126 Quality Model (Source [63])	46
Figure 3-5: General Concepts of the ISO/IEC 9126-1	47
Figure 3-6: Requirement Types as Defined by the IEEE Std. 830	50
Figure 4-1: From Perceived Reality to Model Representation	55
Figure 4-2: Domain and Modeling Ontology and Their Dependencies on Our Work	57
Figure 4-3: Reality, Model, and General (or meta) Concepts	58
Figure 4-4: Component and Process as Fundamental Entities in the Perception of Reality	62
Figure 4-5: Concrete Components and their Properties	67
Figure 4-6: Conceptual Model of Properties of Concrete Components	68
Figure 4-7: Basic Types of Systems	70
Figure 4-8: System as a Special Kind of Component	70
Figure 4-9: Informal Illustration of the Fundamental Concepts of a Concrete System	72
Figure 4-10: Conceptual Model of the Structural Concepts of a Concrete System (incl. LST References)	75
Figure 4-11: Informal Illustration of The Three Fundamental Types of Systems and their Relationship to Software Quality Domains	78
Figure 4-12: Conceptual Model of a Concrete System	80
Figure 4-13: System Models in Section 4.3 and their Informal Relationships	81
Figure 4-14: Conceptual vs. Physical Decomposition of a System	83
Figure 4-15: Conceptual Model of Concrete Systems and their Properties	85
Figure 5-1: Two Relevant System Layers with “Goals-Means” Viewpoints	88
Figure 5-2: Informal Illustration of the 2-2-2 Model of Concrete Systems	89

Figure 5-3: Sequence of Activities to Identifying Properties	90
Figure 5-4: Different Types of Property Decompositions.....	94
Figure 5-5: Collaboration-Based Realization of a Functional Property	99
Figure 5-6: QoS _y as a Cross-Cutting Concern of Collaborations	101
Figure 5-7: Collaboration b with the constraint to fulfill QoS _y (a) is broken into two collaborations (b)	102
Figure 5-8: Distributing Responsibilities Yields New Roles	102
Figure 5-9: Conceptual Model of a System Description	105
Figure 5-10: One possible Notation for a ConcreteSystem.....	107
Figure 5-11: UML Metamodel for the Concept of a TaggedValue (Source [96])	109
Figure 5-12: Property Trace Types as Stereotypes of UML Abstraction.....	111
Figure 5-13: Collaborating Components in the Universe of Discourse	112
Figure 5-14: UML-Based Trace for a Physical System-Level Property	114
Figure 5-15: A Functional Property Tracing to its Realizing Collaboration.....	115
Figure 5-16: UML Sequence Diagram to Represent a Process and Determine the Functional Properties of Involved Components	115
Figure 5-17: An Extra-Functional Property Tracing to its Realizing Collaboration.....	116
Figure 5-18: Refinement of the “Authentication” Collaboration and Integration with “CurrentBalanceDialog”	117
Part III Overview Figure: Informal Relationships Among Key Models and Their Concepts of Part II and Part III	123
Figure 6-1: Informal Illustration of the Quality Model Construction Framework and its Derivation from the Basic Model of Components and Properties (Figure 4-4)	127
Figure 6-2: Extending the ISO 9126 Quality Model	132
Figure 7-1: Specification Levels and Aspects of Business Components (Source [46])	137
Figure 7-2: A UML Metamodel for the Quality Description of a Software Component	138
Figure 7-3: Conceptual and Physical Layers of the RAS.....	146
Figure 7-4: Top-Level of RAS Core Metamodel (Source [116]).....	148
Figure 7-5: Key Extensions of RAS Concepts to Describe Quality Attributes of Software Components	150
Figure 7-6: The AIP Application Paradigm	154
Figure 7-7: Main Application Programming Concepts in the AIP.....	155
Figure B- 1: RAS Core Metamodel of the Classification Part	180
Figure B- 2: RAS Core XML Schema of the Classification Part.....	181
Figure B- 3: RAS Profile XML Schema Based on the Running Example.....	181
Figure B- 4: Example RAS XML Document Based on the Running Example.....	182
Figure B- 5: RAS Profile XML Schema Based on Alternative One	183

Figure B- 6: Example RAS XML Document of Alternative One	183
Figure B- 7: RAS Profile XML Schema Based on Alternative Two	184
Figure B- 8: Example RAS XML Document of Alternative Two	184
Figure B- 9: RAS Profile XML Schema Based on Alternative Three	185
Figure B- 10: Example RAS XML Document of Alternative Three	185
Figure C- 1: Complete Meta-Model of System and Properties	187

List of Tables

Table 1: Examples of Systems and Their Relative Complexity (Source [143]).....	17
Table 2: OOSEM Activities and Main Model Artifacts.....	28
Table 3: Comparison of System Engineering Relevant Features.....	37
Table 4: Requirement Types as Defined by the IEEE Std. 1233.....	42
Table 5: Non-Functional Properties Observable Over Distinct Life Cycle Phases.....	44
Table 6: ISO/IEC 9126-1 Definition of Characteristics and Subcharacteristics for Internal <i>and</i> External Quality (cf. Figure 3-5).....	48
Table 7: ISO/IEC 9126-1 Quality In Use Characteristics (cf. Figure 3-5).....	49
Table 8: Mapping of Miller’s Generic Systems Definition to Concrete Systems [88].....	71
Table 9: Mapping of Miller’s Generic Systems Definition to Conceptual Systems [88].....	77
Table 10: Generic Stakeholder Classification Layout.....	91
Table 11: Stakeholders of an e-Commerce Application for a Supermarket.....	92
Table 12: Property Realization Foci.....	96
Table 13. COM+ Services and Their Support for Quality Requirements.....	100
Table 14: UML Profile for Modeling Hierarchical Systems and Properties.....	106
Table 15: UML Meta-Model Extension for ConcreteSystem.....	108
Table 16: UML Stereotypes for Property Realizations.....	111
Table 17: Example Use of Terminology.....	126
Table 18: Example Possibility Matrix.....	131
Table 19: Asset Consumer Concerns and Solution Approaches.....	135
Table 20: Deriving Quality Attributes from Stakeholder Goals Using the GQM Paradigm.....	140
Table 21: Suggested Quality Attributes to be Published in a Software Component Data Sheet.....	142
Table 22: Potential Future Work: Combining Results of the Thesis Work with Existing Development Methods.....	163
Table 23: Glossary of General Terms.....	172
Table 24: List of Abbreviations.....	174
Table 25: Informal Definition of High-Level, Non-Functional Properties.....	175

1 Introduction

Objectives: This Section shall help the reader to gain an overview over this thesis by providing a description of its motivation derived from software- and systems engineering practice and its underlying research problem(s). This Section will also briefly present our solution approach and an explicit statement about our main contributions.

After having read this chapter, the reader will:

- know that our basic motivation is based on the immaturity of current development methods with their methodological support for dealing with non-functional properties of systems, and for dealing with hierarchies of systems;
- know that we strived for a more fundamental understanding of the generic nature of systems and of properties in order to suggest generalized methodological support for the diversity of systems and properties to be considered during systems development;
- be informed about what we believe to be the major contributions of this work;
- know what to expect of the forthcoming Chapters of this thesis and how they relate.

1.1 Motivation

As Simon stated in his book *The Sciences of the Artificial* [126], whose title shall allude to a science of engineering: "...artificial objects – and more specifically prospective artificial objects having desired properties – are the central objective of engineering activity..." By *artificial* Simon means man-made as opposed to natural. In the case of systems engineering, these artificial objects are systems; in the case of software engineering, they are software. When we discuss artificial objects in engineering, we do so in terms of their properties. Properties are used in an imperative or descriptive style. In more engineering related terms, we use the properties for the purpose of expressing requirements on the object to be designed ("to-be-built" properties) and for the purpose of expressing the result, i.e. the finally exhibited properties ("as-built"). In addition, any engineering work is usually concerned with analyzing some existing object. We therefore also use properties to express an "as-is" situation.

Our motivation for conducting the research work described in this thesis is grounded in the lack of a general theory on what exactly these desired properties are, i.e. what principal types of properties there are and of what general nature they are. We, as engineers, need such an understanding in order to be able to develop general approaches on how to identify the relevant properties in a given situation, and on how to design such artificial objects that then shall exhibit the identified properties. Today, the design of any non-trivial system is a teamwork effort. This requires that we be able to communicate our "as-is", "to-be-built", and "as-built" properties of systems by means of representations that are as unambiguous as possible. Because properties cannot be discussed independently of the objects they are ascribed to, any such conceptual framework must be explicit about the type of objects and types of properties considered. We limit our domain of design to software-intensive systems. Software-intensive systems are systems in which software plays a key role in delivering its overall functionality.

Consequently, the relevant engineering disciplines are systems- and software engineering and the relevant objects are software-intensive *systems*, their environments, and their constituents.

The lack of a general theory on properties of systems, which is applicable to the systematic design of man-made systems, is surfacing in practical issues encountered in systems- and software engineering today.

For every newly introduced type of object, e.g. the object of a Web service to build Web-based applications, or the object of a software component to build GUI applications, there is usually no consensus as to what properties are to be considered for describing the object sufficiently. This is especially true for the vaguely defined non-functional properties. Not only is there a lack of consensus on *what* properties are to be considered but also do we lack a general, i.e. an object-type independent, methodology on *how* to approach finding the relevant set of properties. The general problem manifests in diverse so-called quality models or quality attribute classifications, confusing terminology, etc. This in turn makes it hard for potentially interested parties to properly evaluate a certain object for its suitability based on its described, or not described because not considered, “as-built” properties. For this very reason, the notions of component data sheets, quality contracts, or component quality models have recently come up in research as a means towards a general model on how to describe properties of software components. These properties are particularly important for software components because the latter are meant to be parts, even third party parts, from which software systems are assembled. And because the system properties are largely determined also by the component properties, there is in general a need to standardize on common structures and formats to describe properties of reusable assets such as software components. The Reusable Asset Specification [116] is such a recommendation but does not yet cope with software components and neither define a systematic way to cover non-functional properties.

System and property modeling in current development methods face shortcomings, particularly in three interrelated areas:

- (1) Support for *systems* development: lack of conceptual foundations to cope with hierarchical systems.
- (2) Support for *non-functional properties*: lack of conceptual foundations to cope with more than functional properties.
- (3) *Customizability* of development methods: lack of conceptual foundations to support the customization of development methods so as to be able to cope with hierarchical systems and non-functional properties.

1.1.1 Support for Systems Development

These days, practicing software engineers are faced with the fact that software development entails more than the production of single, well-defined software applications. It rather stands for the realization of a business objective by a careful consideration and alignment of several resources (people, hardware, software). On a high level, systems development is therefore different from software development in that:

- not only information but also matter and energy processing is considered;
- people and other physical entities are equally important, integral components that take over well-defined responsibilities;
- and frequently, several IT-based systems are integrated into a whole. Each of them might be a system on its own right providing its own applications. Hence, IT-based systems become components themselves.

To cope with this challenge of *systems* engineering as opposed to “software engineering only”, engineers are faced with a spectrum of methods involved in developing such systems. This may range

from: (a) A combination of requirements engineering methods blended with business modeling, software architecture approaches, and software development methods with a resulting heterogeneous set of modeling approaches. (b) A single *software* development method that provides in addition to its software focus minimal support for contextual modeling (also called problem domain or business modeling). (c) A single software development method that provides no such support and hence leaves the software engineers with ad hoc approaches to structure the knowledge needed to capture the full systems scope. Whatever be the case, all these methods have their own underlying conceptualization and basic philosophical take on systems and properties. This burdens their combined use in that overlapping and redundant concepts are present. Further, because today's software methods original focus was on *software only*, which amounts to information processing only and thus covers a certain type of properties only, they do not provide for the generality needed to address systems engineering.

However, a successful combination of business processes with the proper employment of human resources and information technology that is based on reusable and modifiable software (hence, the strive for software components) is key to the cost-efficient achievement of business objectives. A central desire on a *systems* development method would therefore be its (model-based) support for a natural and traceable progression through *hierarchies of systems* - bluntly speaking "from business to java classes"; i.e. from a market environment over the business organization to the responsibilities taken over by the various parts of the technical system to be developed, ultimately of course also by the software parts. The traceability issue cannot be stressed enough considering the frequent changes in our environment and the resulting need to change software or to change business processes as a result of deployed or even changing software.

1.1.2 Support For Non-Functional Properties

The invading pervasiveness of Information Technology (IT) in our products and processes calls for a continuous development of classical and also unprecedented software-intensive systems. It is a truism that in addition to the diversity, the complexity of the contained software as well as the creation thereof is increasing. Besides the pressure on low software production costs, this complexity is caused by the need for increased functionality per application, the growing number of applications with an increasing level of application integration, and the greater diversity of properties to be considered through the earlier-mentioned systems focus. Consequently, this leads to the realization that *more* than the barebones functionality is *required* from the software-intensive system to be produced. This last point is frequently subsumed under the somewhat fuzzy term *non-functional requirement*. The notion of non-functional properties has a specific connotation in the software engineering community. It became to stand for all desired or exhibited properties that exceed a software-intensive system's main input/output behavior in its execution environment; i.e. they exceed the system's main functions. Consequently, non-functional properties cover a rather diverse range of phenomena embracing both *execution-related properties* in the deployment environment of the software (with high-level properties such as security, performance, etc.) and the software *development-related properties* (with properties such as maintainability, reusability, etc.). While the former have an influence on the economics of the company executing and utilizing the software, the latter have an influence on the economics of the company developing the software. The increasing focus on such properties is in particular the result of two major developments:

- *Execution-related properties*: Software became pervasive and thus also key in business-critical functions or, in general, in functions that humans rely on in their everyday life. Consequently, it is not only important that software does provide certain functionality but also *how* it provides this functionality.
- *Development-related properties*: Because producing and selling software shall be a viable business, pressure on fast delivery and low cost software production and maintenance impacts the way we are supposed to create such high-quality software. Hence, the properties related to improved maintainability and reusability are given higher priority

due to the awareness that maintenance efforts are far exceeding the development efforts of software systems [49] and that the reuse of software artifacts is only advisable for high quality artifacts. The trend towards assemblage of software systems from software building blocks, be they newly developed or pre-existing, is a result of the “faster-better-cheaper” vision. Such systems are increasingly developed following a software component-based approach [131] or even a product line paradigm [30]. However, predicting properties of assemblies requires knowledge of the properties of the constituent components and their composition arrangements. This realization raised the awareness for having functional and non-functional properties of such software components made explicit [35, 125].

Software development methods historically concentrated on systematic approaches to realize functional requirements but left the non-functional requirements aside.

The non-functional properties, which relate to the development economics, were largely seen as the responsibility of software quality management, which is only loosely, if at all, connected to the software development methods. Consequently, methodological guidance for such properties is often relegated to an assessment of the static qualities of already produced software artifacts (e.g. source code characteristics such as the number of methods per class, etc.), or they are meant to be positively influenced by more rigorous development processes and control mechanisms (e.g. by well-defined intermediate artifacts to be produced and reviewed). Further, programming paradigms, such as structured programming or object-oriented programming, were in general perceived to be conducive to software quality.

The non-functional properties, which relate to the execution characteristics, were treated as a technology issue related to the deployment phase and as such again outside the scope of development methods.

As a result of the current state, the practicing software engineer, who relies on development methods as the prime vehicle to guide his engineering efforts, is not adequately supported in coherently dealing with the majority of properties that the end result has to exhibit. The still emerging field of software architecture [14], once it will seamlessly be integrated into software development methods, could become a remedy for some types of non-functional properties

1.1.3 Customizability of Development Methods

Our practical experience in developing large-scale software for industrial products showed that a software development method is hardly ever used as defined by the method’s authors. Companies have to modify methods and combine bits and pieces from different methods with their own in-house practices and add-ons. This statement is validated by recent surveys in the UK (referenced and summarized in [8]), where only 11% of the companies using a software development method reported the use of an unmodified “commercial” one.

One of the shortcomings of current development methods is their inadequacy for developing and thus modeling systems of systems. The inadequacy is based on mainly two aspects:

- For many methods, the underlying assumption is basically a two-level approach to modeling of *one* system: system context (e.g. relevant business environment with system users) and *the* system, whose internals are essentially software artifacts. In object-oriented approaches the latter are objects, in component-based approaches the latter are software components, and in architecture-first approaches the latter are architectural, i.e. conceptual software components. Because of this implicit restriction, the methods introduce level-specific, focused concepts and principles that make it unnatural and complicated to be applied to scenarios where the system under consideration is composed of more than these two levels, such as when several IT-systems make up the system to be

developed. Hence, the introduction of only one more level becomes an unforeseen aspect that needs customization.

- As also the authors of Kobra observed ([4] p.61), many methods introduce multiple similar concepts for the same underlying idea. For instance, a use case is used to specify functional requirements at system level, but not to specify functional requirements at component level (where for example something like Fusion's [33] operation schemata might be used). Such kinds of concept redundancy introduce hard to manage complexity especially in those cases where several levels of systems are to be developed, because we might end up having specific concepts for components on every level despite the same, but implicit, underlying idea of what needs to be modeled.

Customizing methods requires that the underlying foundations, often called philosophical underpinnings, be explicit. By underlying foundations we refer to the foundations and assumptions behind both the development process and behind the employed systems modeling approaches. For example, we mean explicit statements on the kinds and nature of systems and properties the method can and cannot deal with, at what stages ingenuity is required versus stages where a mechanical processing is possible, what assumptions and constraints its application entail, etc.

Only if these aspects are made explicit is it possible to infer why a method functions the way it does, in what cases it does not work, what the nature of the problem is it can help solving, and what parts can safely (with understanding of its impact) be replaced, modified, extended, or even omitted. All this is needed to do "method engineering", i.e. to create new or modified methods based on existing methodological components.

1.2 Research Problem

The research problem is grounded in the conception of the basic methodological building blocks and fundamental underpinnings that would allow development methods and their modeling paradigms to eventually cope with the greater diversity of systems and greater diversity of properties (a consequence of Sections 1.1.1 and 1.1.2) and that would provide explicit philosophical underpinnings that facilitated method engineering (Section 1.1.3). This raises the question of suitably generic system and property abstractions and consequently required a more holistic understanding of both the nature of the systems and the nature of the properties to be considered. Therefore, the research problem space can be divided into two clusters:

- (1) Develop a fundamental theory about systems and properties. Its contents shall help to develop a more generalized understanding about, or holistic view of, the various systems and properties that need to be understood (and thus expressed in models) as part of the system development life-cycle.
- (2) Based on fundamental insights gained from the first cluster, find methodological improvements, if needed entirely new methods, to address problems in current development practice related to multi-system and multi-property modeling.

As practical constraints we foresee that such methodological building blocks would be taken into consideration the current model-driven emphasis of development methods and preferably leverage on existing ideas. Further, they shall prove useful in bridging the gap from systems- to software engineering (in particular component-based software engineering) in that for concept uniformity is striven whenever possible. The practical constraints are reflected in the topics that are discussed in Part I – the Context of this thesis.

It originally appeared to us that a coherent basic theory of non-functional properties is akin to what Checkland coined a 'fragmented adhocacy' [28]. It refers to any intellectual field where some state of confusion is the result of technological and practical advances that outrun the development of thinking (theory). This state of confusion was also observed by others ([14] p.423 or [17] [55]).

Because one of our goals is to be able to conceptually approach as many properties as possible in a uniform way, we wanted to understand whether there are classes of properties that have fundamentally different natures and therefore would justify fundamentally different methodological approaches to cope with them. To facilitate a conceptual cleansing, we had very basic questions for which we pursued answers:

- What are properties (or quality attributes as they are called in software engineering)?
- What do we ascribe properties to?
- What distinguishes, and why do we as software engineers distinguish, functional from non-functional properties?
- Is there a more general theory or model to conceptualize the wide range of properties?

Because our properties of interest are usually predications of systems and because we want to be able to cope with more than software systems, the inevitable second set of questions revolves around the notion of systems:

- What are systems?
- What basic kinds of systems and their interrelationships are we dealing with in software engineering?

Finally, since both perception of and reasoning about properties and systems are fundamentally cognitive processes, we also wanted to know how we perceive systems and how we construct mental models of them. These insights shall be influential for deriving “human-compatible” recommendations on modeling approaches.

1.3 Solution Approach

Any methodological support for the development of systems, i.e. for the design of an object with desired properties, requires a theory of systems that is general enough to cater for the diversity of systems to be considered. In this thesis we offer such a general theory for a more holistic understanding of systems and properties, with a practicality emphasis on software-intensive systems. The conceptual model depicted in Figure 1-1 is at the heart of the theory. The figure is a simplified version of the model we derive in this thesis (thus “Partial” in the figure caption) but it serves the purpose of explaining our solution approach.

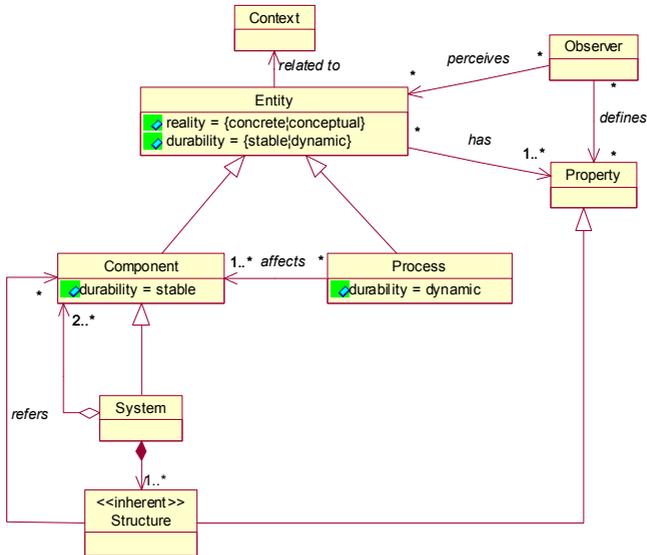


Figure 1-1: A (Partial) General Model for Systems

On a high level, our model is founded on the combination of the following three major building blocks and its implications:

1. Concrete and conceptual components and processes

In our supposition of the nature of our conceptions about entities in the world, we distinguish between *concrete* entities¹ and *conceptual* entities (indicated through the “reality” attribute of Entity in Figure 1-1). While the former refers to something in the real, physical world, conceptual entities refer to something that is conceived in mind. Conceptual entities are used to handle knowledge. In order to not remain tacit knowledge, i.e. entities internalized in a single mind, conceptual entities must be externalized and thereby represented in some form; for short, they become representations.

Entities can be *components* or *processes*, i.e. something relatively stable and something that is perceived as dynamic, respectively. This is indicated through the “durability” attribute of Entity in Figure 1-1. As a consequence of our “reality” attribute, there exist concrete components and processes as well as conceptual components and processes. Concrete processes amount to some change of matter or energy in concrete components; conceptual processes amount to some change of knowledge/knowledge representation in conceptual components. In other words, conceptual processes refer to “pure” information processing. Thus, processes change components. This change is expressed through some of the properties of components.

2. Concrete and conceptual systems

A system is a component whose structure is of interest to us. Therefore, having a structure is an inherent or intrinsic property of a system. By inherent we mean that a system always has this property

¹ Entity: something that has separate and distinct existence and objective or conceptual reality [84].

independently of its relationships to other things. The structure, referring to the arrangement of components, together with our natural perception of phenomena makes any concrete system of further interest a hierarchic system. Thus, our modeling metaphors and constructs need to take this into account.

Any *concrete system* that is of interest to us, such as a company, a human, an IT-based information system, etc. has a *behavior template*, i.e. a first information input that serves as the starting point for its later behavior and structure. This is the genetic information in the case of humans, but more importantly in the case of software-intensive systems, the behavior template is the *software*, a *conceptual system*, which is the first input information for the computer system.

3. Concrete and conceptual properties

To both components and processes we ascribe properties. Consequently, we have properties for concrete and conceptual components and for concrete and conceptual processes. Our research suggests four basic types of properties for concrete systems: functional properties, extra-functional properties, physical properties, and conceptual properties. The first three types of properties relate to the system or its parts being perceived in its operational context. Conceptual properties are either properties of the behavior template or they are properties ascribed to the system because it is seen in any other conceptual context, i.e. in any context that serves to structure knowledge. For example, such a context could be a categorization scheme such as “all systems that were designed by Leonardo da Vinci”, or “all men who are taller than 2m”, etc.

In summary, our basic concepts to describe concrete and conceptual reality in the context of systems are components, processes, and properties. Systems are special kinds of components in that we are interested in their internal structure. This is so, because either we want to find out how a system functions or we want to design a system that has to function in a certain way.

Based on this model, we can now better describe other pieces of our solution approach. These pieces are either conceptual building blocks, in an attempt to help solving the problems/questions mentioned in Section 1.2, or applications of these building blocks that amount to partial solutions to the practical problems mentioned in Section 1.1. In Figure 1-2, we illustrate the pieces with dotted ellipses. Those of conceptual nature have legend numbers C1, C2, etc. and those of applied nature A1, A2 and so forth.

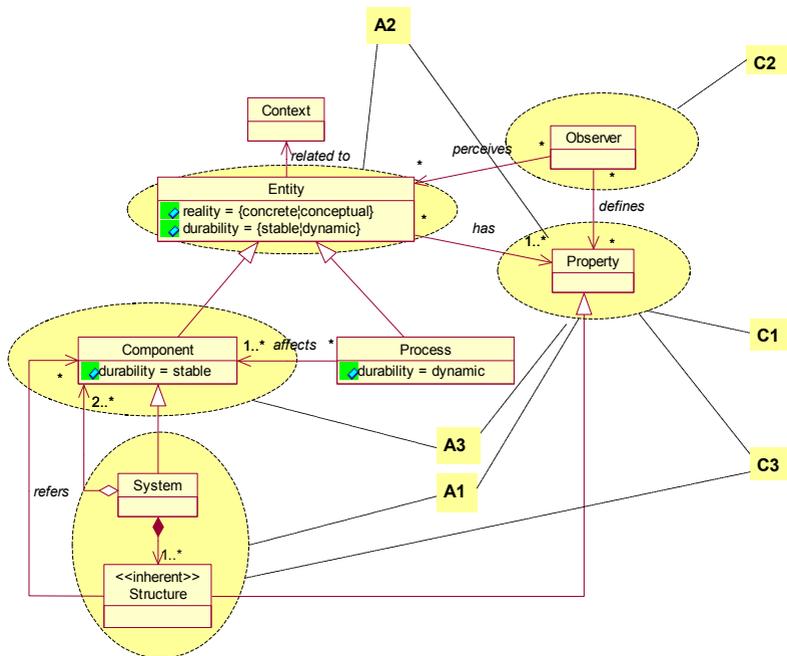


Figure 1-2: Pieces of Solution Approach

Besides deriving the basic model, as discussed above, and stating its implications and assumptions for systems- and software engineering, our solution approach encompasses the following (listed in the order of discussion in this thesis):

- C1. *Fundamental types of properties*: A basic account and our derived model of the fundamentally different types of properties of concrete systems is the basis for most other work described in this thesis, i.e. for most other building blocks in our solution approach. For this purpose we combine our interpretations of the research conducted in basic philosophy and cognitive science.
- C2. *Stakeholder identification/classification*: Since the observer is the all-important component that defines what properties are considered and what their meaning is, we developed our 2-2-2 model, among others applicable to more systematically identify and classify the relevant stakeholders for a system. This model uses the dimensions of life-cycle based context, goals/means, and hierarchical system level to group classes of stakeholders.
- C3. *Property traceability through hierarchies of systems*: Because our systems of interest are inherently hierarchic, i.e. recursive decompositions of concrete systems, we propose a method on how to conceptually trace the fundamentally different properties through such hierarchies of systems.

- A1. *Representation of systems and properties in the UML*: Because the development of any complicated system is usually a team effort, we need transferable representations of our knowledge. For this purpose, we suggest how to represent our conceptualizations of systems and their properties, as well as property traceability, in the UML.
- A2. *Quality attribute construction model*: Any property is only meaningfully definable if the entity, its context, and its observer is defined. For this purpose, we suggest a construction method for property models of entities. This construction model represents an extension of the quality model suggested by the ISO/IEC 9126 standard. The extension essentially brings flexibility and a systematic approach to the definition of measurable quality attributes for any entity in question.
- A3. *Software component data sheet*: As an application of the quality construction model and as an application of the idea of viewing a software component as both a concrete and a conceptual system, we suggest a model on how to describe “as-built” properties of software components. We show how such a quality description can be incorporated in an extended version of the Reusable Asset Specification [116].

1.4 Research Method

Our method of research was biased by the way research in systems science is conducted: *systems inquiry* [10]. Systems inquiry is itself a system - a conceptual system - and defined via four interrelated aspects (see Figure 1-3): System philosophy and system theory are used to produce systemic knowledge, system methodology together with system application represent systemic action. This framework can be adopted in a conclusion-oriented or decision-oriented mode. While conclusion-oriented refers to producing more systems knowledge, decision-oriented refers to applying systems knowledge to the formulation and selection of methods that address real-world problems.

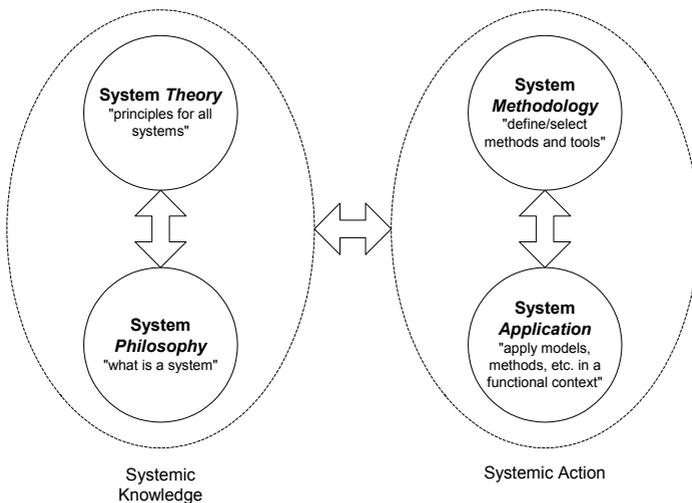


Figure 1-3: Systems Inquiry as Proposed by Banathy [10]

How did we apply this framework? In general, we used this framework in a conclusion-oriented mode to produce property-related knowledge and combine this with existing systems knowledge. This

combination of knowledge – amounting to our theory of systems and their properties - is then used in a decision-oriented mode to formulate methodological building blocks for the modeling of systems and their properties, and for applying these building blocks to address real-world problems in systems- and software engineering. Our research maps on the individual aspects of systems inquiry as follows:

- **Philosophy aspect:** We researched basic philosophy and cognitive science (a field drawing upon philosophy) to understand *what are properties of systems* (ontology) and *how do we perceive and conceptualize systems and their properties* (epistemology). We adopt a conclusion-oriented mode in that we formulate the gained knowledge as basic tenets that shall help to attain an attitude of mind suitable to conceptualize properties and systems. We also adopt a conclusion-oriented mode when we interpret our findings to explain the pervasive notion of a non-functional property.
- **Theory aspect:** We define common concepts and principles for generic types of systems and their related properties. The concepts and concept relationships amount to a holistic conceptual model of systems and properties, and as such can be viewed as a meta-model of a modeling ontology. By this, we try to answer what commonalities do certain types of systems and certain types of properties, which we could use to approach dealing with them in more uniform ways.
- **Methodology aspect:** Based on the gained knowledge, we propose methodological building blocks that can be incorporated into existing development methods. These building blocks are partly inherent in the tenets and partly explicit such as the 2-2-2 model, as the basic trace patterns of properties through realization hierarchies of systems, and as the suggestion on how to construct a context-sensitive, customizable quality model as an extension to the ISO/IEC 9126 quality model for software products.
- **Application aspect:** We use the produced new knowledge and the methodological building blocks to address current shortcomings in systems- and software engineering, as were mentioned in the motivation Section 1.1. For instance, we suggest an approach to represent system hierarchies and property traces in UML-based model representations; we identify the relevant stakeholders (which is a prerequisite to identify the set of relevant properties) by means of the 2-2-2 model, and we propose a quality attribute description for software components.

1.5 Contribution

We structure the description of our contributions by again using the systems inquiry framework. Consequently, this thesis provides its contributions to (1) *knowledge* and (2) *action* in systems- and software engineering:

1. Contributions of ontological and epistemological nature, which help to understand systems and properties on a more abstract but holistic level, and therefore support a conceptual cleansing in the area of systems and non-functional properties. Concretely, we contribute by:
 - Suggestions on how non-functional properties can be understood and classified given the basic philosophical and cognitive definitions of properties. This includes making explicit the basic concepts that need to be considered when properties of concrete systems are to be conceptualized.
 - Reinterpretation of definitions and terminology around the relatively fuzzy field of non-functional properties.
 - Reemphasizing and proving from a philosophical, cognitive, and systemic standpoint why hierarchies of systems are the most natural way to model systems.

- A general model of systems and their properties, including the definition of the fundamental types of properties of concrete systems. The model may serve as a concept model for systems in systems engineering.
 - Formulating recommendations and insights in concrete tenets that help to build an attitude of mind for the purpose of conceptualizing systems and properties.
 - The relationship of the three important types of systems to the quality realms in software engineering.
2. Contributions of methodological and applied nature, which have an immediate practical value:
- The 2-2-2 model as a means to put a software system into its important contexts to clearly address a certain class of relevant properties.
 - An application of the 2-2-2 model to discover and classify stakeholders.
 - A principled manner to trace the fundamentally different types of properties through hierarchies of systems.
 - A suggestion on how to represent properties and property traceability in the UML.
 - A flexible quality attribute construction model for software products. It can be applied to identify and introduce measurable attributes and their metrics in a ISO/IEC 9126 compliant way.
 - A conceptual model for expressing quality attributes in a software component data sheet together with a list of suggested quality attributes.
 - An extension of the logical model of the Reusable Asset Specification (RAS [116]) to accommodate quality descriptors. This allows representing the quality attributes of a software component data sheet in a standardized way.
 - A suggestion for an XML-based physical model of the RAS, which is amendable to advanced tool support in that the compliance with the logical model can be verified.

1.6 Roadmap

The report is organized into Parts, Chapters, and Sections within Chapters. The core Chapters of this report (Chapters 2 to 7) are organized into three Parts as depicted in Figure 1-4. The lines among the Chapters or Parts indicate their dependencies, in that a Chapter is influenced (arrow head) by the information generated in another Chapter. The Introduction and Conclusion Chapters are related to all other Parts, which is why we dispensed with showing them.

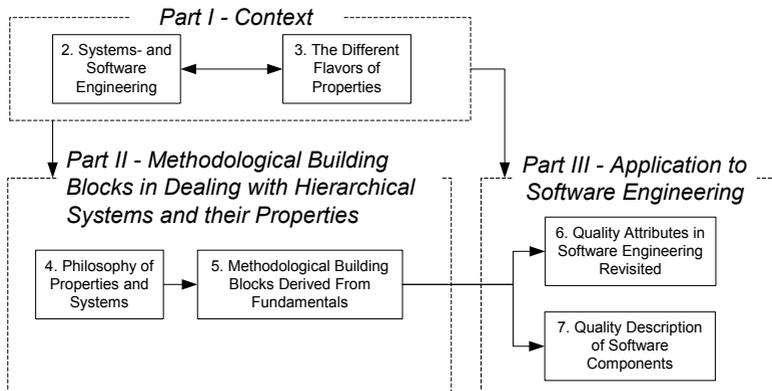


Figure 1-4: Chapter Layout and Influence Relationships

The rest of the thesis report is structured as following:

Part I: Contains the Chapters that provide the relevant context for our work. As such it introduces model-based development methods of systems- and software engineering and the current understanding of properties in these disciplines.

Part I – Chapter 2: Positions the work reported in this thesis in the context of systems- and software engineering. In particular, we give an overview of existing model-based systems- and software development methods and their approach to modeling hierarchical systems and especially to modeling different types of properties. We list the basic requirements on an envisioned systems modeling language and contrast these requirements with the features provide by the modeling approaches of the presented development methods. In addition, a brief introduction of extension mechanisms of the Unified Modeling Language is given.

Part I – Chapter 3: Gives an overview of the current understanding of the notion of properties and its synonymous terms in systems- and software engineering. In order to give an account of the understanding that is representative for the fields of systems- and software engineering we discuss several international standards. Further, we explain why the notion of quality (either standing alone or in combination with attribute or property) has a special connotation in software engineering.

Part II: Contains the Chapters that discuss the pieces that together form our general, conceptual framework as an aid to more holistically understand hierarchical systems and properties and to use this understanding in coping with properties during systems development.

Part II – Chapter 4: Presents our basic conceptual framework for the general understanding of properties and of systems. After a brief introduction into the fundamentals of modeling, it reports on our investigations of the fundamental nature of properties and of systems. The investigations cover

basic philosophical and cognitive aspects of properties and the conceptualization thereof. Our conclusions derived from this research are formulated in concrete tenets that we hypothesize to be important in any methodological approach that is intended to cope with a large diversity of properties of systems. Further, our conclusions result in a holistic, conceptual model of systems and properties.

Part II – Chapter 5: Builds upon the philosophy derived in Chapter 4 in that it shows how the provided knowledge is combined into methodological building blocks that could be integrated into conceptual frameworks for systems development. Chapter 5 presents the 2-2-2 model. This model suggests three orthogonal dimensions, from which a system under consideration can be viewed at. Intersections of these dimensions hint at interesting perspectives of a system. Each perspective is related to certain types of stakeholders and therefore certain types of properties. Chapter 5 also presents how the fundamentally different types of properties can be traced in a principled manner through realization hierarchies and how systems, properties, and property traces can be modeled and represented in the UML.

Part III: Contains the Chapters that show concrete applications of the conceptual framework in order to partially contribute to the solution of pending problems in software engineering.

Part III – Chapter 6: Revisits the prevalent definitions of quality attributes and its related terms in software engineering and suggests a core terminology based on our work. The key part of Chapter 6 is the presentation of a quality model construction framework. It is intended to be a methodological approach to constructing customizable, context-sensitive quality models that help identify and classify quality-carrying attributes for given entities in defined contexts. Chapter 6 presents how the quality model framework can be used to improve the ISO9126-1 meta-model.

Part III – Chapter 7: Presents a conceptual model to describe quality attributes of software artifacts, in particular suited to software components. The quality description – a metamodel for a component data sheet – draws upon the systems theoretic findings presented in Chapter 4 and the quality model defined in Chapter 6. Chapter 7 explains how the metamodel of the Reusable Asset Specification (RAS [116]) can be extended to incorporate such quality descriptions. It delegates to Appendix B the discussion on the possible alternatives to represent the RAS metamodel in the form of XML Schemas to facilitate different degrees of machine-verifiable meta-model compliance. Finally, Chapter 7 shows how we used the RAS to describe ABB-specific software assets.

Chapter 8: Chapter 8, the Conclusion, summarizes the results of this work. It discusses paths for future work with respect to both further research and the application of the presented ideas in practice.

Part I

Context

Our work is interdisciplinary. It has an emphasis on a holistic understanding of systems and properties and its value in general systems modeling. It aims to support software-intensive systems engineering, which is on the borderline between systems- and software engineering. Part I therefore introduces systems engineering and software engineering, the special sub-discipline of component-based software engineering, and the notion of development methods. The most prominent representatives of systems- and software development methods are briefly reviewed with respect to their approaches to modeling properties of systems and hierarchies of systems. Further, the UML and its extension mechanisms are introduced to the extent needed in our work. The second Chapter of Part I discusses the different types of properties that are to be considered in the development of software-intensive systems. We elaborate on the notion of non-functional property, reflecting its current understanding in the systems- and software engineering community. We posit that properties cannot be discussed independently from systems, and that it is largely due to the consideration of the greater diversity of properties and systems that we proceed from the discipline of software- to systems engineering.

Part I provides sufficient background so that the generic discussion of properties and systems, which follows in Part II, is motivated and can be put into context.

2 Systems- and Software Engineering

Objectives: This Chapter shall help the reader to obtain enough contextual information to be able to position this work in the larger scope of systems- and software engineering and in particular in the scope of system- and property modeling as part of these engineering disciplines.

After having read this chapter the reader will:

- be able to position the disciplines of software engineering and systems engineering with respect to each other;
- know our general definition and the objectives of a development method, valid for both systems- and software development;
- have an overview over the most prominent existing systems development methods as well as component-based software development approaches, which could potentially be extended to accommodate a general systems and properties modeling paradigm;
- know about the principle extension mechanisms of the UML, which we need to draw upon, if we want to make use of the UML as a language for general systems and properties modeling.

2.1 The Discipline of Systems Engineering

Systems engineering is an interdisciplinary approach with the promise to provide means to enable the realization of *envisioned systems*. Envisioned system stands for a complicated, *designed* system (i.e. one created by man) as opposed to a natural system. According to the International Council on Systems Engineering [62] “Systems Engineering integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation.” It is the focus on team-effort among specialty groups that requires an extra effort on securing a shared understanding in order to communicate precisely. Basic premises for doing so are, firstly, an agreed upon terminology expressed in a general model of the fundamental concepts and concept relationships that are particular to the systems engineering domain. Secondly, a common modeling language with its defined syntax and semantics shall allow transferring knowledge among different specialists involved in the creation of systems. Consequently, both the elaboration on a conceptual model for systems engineering and the creation of a common modeling language are high on priority in the systems engineering community [97].

Contrary to the specialists in particular fields, a systems engineer is an “intelligent” laypeople, i.e. she/he is interdisciplinary skilled and most interested in acquiring a holistic understanding of a complex phenomena. It is only through this holistic understanding that a systems engineer can decompose the challenge of building a system into interrelated tasks for the specialty areas. Organizing and communicating such an understanding requires models and their representations. To facilitate this undertaking it is a goal of the systems engineering community to make models and model representations more precise and standardized (discussed later in Section 2.5). A large part of the job of the system engineer is done when the responsibilities of the various components related to specialty disciplines have been identified and communicated by means of models.

We consciously defer the discussion of what exactly a system is until Section 4.3. At the moment, we substitute a general definition with a definition by examples (Table 1). In general, however, the

relevant branch of systems engineering for us is the one, where at least parts of systems are realized in the form of computer-based components. But *the* end product of interest is the system with its properties and not solely the software. In an inaugural paper on computer-based systems engineering [143] the authors give examples of software-intensive systems and classify them according to market domain and their relative level of complexity (see Table 1). While the exact metric for the level of complexity was defined very vaguely, the table can still give an intuition for the breadth of types of systems that are relevant in computer-based systems engineering. Virtually all systems above the low level (and today even those) are computer-based. Further, it is not uncommon that high complexity systems have more than one million requirements and it is interesting to note that high complexity systems are usually so because of their desired non-functional, execution-related properties.

Table 1: Examples of Systems and Their Relative Complexity (Source [143])

Complexity	Defense	Commercial	Public	Telecommunications	Automotive	Financial
Very high		Payroll information management			Intelligent highway systems	
↓	Space station	Airline reservation	Air traffic control			Econometric model
Medium	B1 bomber or cruiser	Manufacturing automation		Network control system	Dealer networks	Portfolio Management
↓		Process control system			Engine management	Electronic funds transfer
Low		Manufacturing cell control		LAN control	Cruise control	Automatic Teller Machine (ATM)

We started our motivation for this thesis by referring to Simon and his observation that synthesizing artificial objects with desired properties is at the heart of any engineering discipline. As we mentioned in Section 1.1.1 already, systems engineering is not only concerned with processing of information but also with processing of matter and energy. Consequently and as depicted in Figure 2-1, we can make a high-level, fundamental distinction between systems- and software engineering by highlighting the considered and to be engineered types of things with desired properties. In the case of systems engineering, these are any combination of matter, energy, and information. In the case of the original definition of software engineering, these things are information. A more substantial discussion on this distinction follows later in Section 4.3. in the context of concrete vs. conceptual systems and their processing of matter, energy, and information

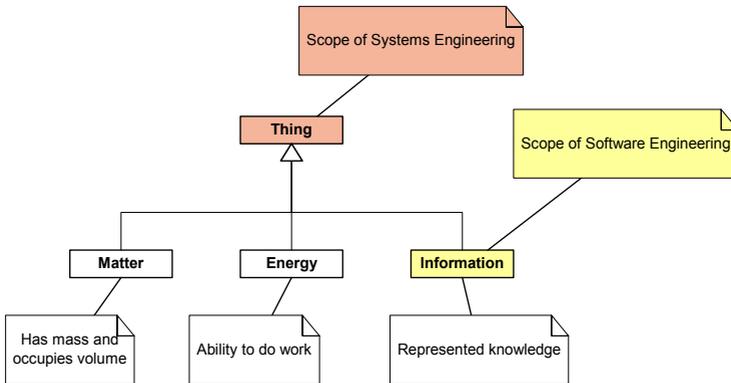


Figure 2-1: Scope of Systems- vs. Software Engineering

2.2 The Discipline of Software Engineering

According to IEEE’s Standard Glossary of Software Engineering Terminology [57] software engineering is both the field of study of systematic approaches and the act of applying them to the development, operation, and maintenance of software. Software is defined very loosely and encompasses “computer programs, procedures, and any associated documentation and data that pertains to the operation of a computer system”. In short, software engineering is the application of engineering to software (and only software). With the citation’s definition of software by example, our scoping of software engineering as illustrated in Figure 2-1 is consistent with the original IEEE terminology.

Work in software engineering can be grossly divided into two major areas: the technical aspects and the managerial aspects. While the former are concerned with the methods and techniques to specify and properly realize specifications in the form of deployable software, the managerial aspects concentrate on all facets to be considered to manage software projects (planning, organization, control, etc.).

Software engineering is more than programming. The essential difference between software engineering and programming is nicely expressed by Parnas who said that software engineering is a “multi-person construction of multi-version software” [103]. Hence, key is to be able to work in teams and to work on software that is gradually evolving. This requires two basic kinds of support: (a) a common understanding and therefore an explicit representation of what needs to be achieved and an aligned process to join forces to achieve what is promised. Both are communicated by means of models: models of the envisioned end product (the software) and its intermediate artifacts, and models of the process by which interrelated artifacts are created. (b) If software shall be made evolvable in a systematic way we must be able to trace evolution increments. Evolution needs are inherent to today’s evolutionary development approaches. But even in non-evolutionary development models, similar needs are called for by new/changing customer requirements, by uncovered failures in the current version of the system, or by external forces (such as technology out-phase). Consequently, if we assume a model-based approach, we require traceability support within and across models.

In spite of IEEE’s original (and focused) definition of software engineering, the current understanding in practice and in research attributes more to software engineering than the design of software alone. Because software is only one, albeit an increasingly more important, part of complex systems made up of hardware, software, and specific environments, software engineering has fluent boundaries to (and is often even synonymously taken for) the discipline of systems engineering and in

particular to computer-based systems engineering [128] [143]. Other authors from the software engineering community make an explicit distinction between software- and systems engineering in that the latter is supposed to put software into context [112]. But, the methods for software development should have fluent boundaries to those methods that are used for the context. Compared with “pure” software engineering, systems engineering addresses a greater diversity of properties (incidentally those that we call non-functional properties in traditional software engineering) and a greater diversity of systems (i.e. an equal emphasis is put on the environments of the software). In the context of computer-based systems engineering, *systems* stand for software-intensive systems that range from enterprise information systems over process control systems to embedded systems. Software-intensive systems engineering stands for the objective to fulfill a purpose with a system of collaborating resources: usually people, hardware, and software. Hence, software is to be seen as an integral part of a larger assembly of parts. This creeping enlargement of the scope of software engineering towards systems engineering is the reason why so-called software development methods increasingly suggest activities and models that make some provisions for taking this wider scope into consideration. For example, several methods include features to model the configuration of physical devices and their included application packages (e.g. locality model in RUP SE [26] or the notion of “physical view” in software architecture [71]). Further, so-called analysis models frequently include the modeling of business processes and identify the roles to be taken over by software [29]. However, the modeling approaches for the non-software parts are different in that often other conceptual frameworks and notations are used. Consequently, the transition from non-software to software models is not an easy one.

In this thesis we use the term systems engineering especially in those occasions, where we do not want our discourse to be understood in the “narrow” sense of software only.

2.2.1 Software Architecture

In this section we briefly introduce software architecture as the principled study of the overall structure of software systems [14, 61]. We do so, because software architecture is the one field of study in software engineering whose justification is closely related to the issues of non-functional properties.

The field of software architecture emerged as a separate field of study from the need to describe software organizations of complex software systems. Complex software systems are those that cannot simply be realized by one person. While theory lacked behind practice for many years, i.e. software architecture tried to capture what is practiced in such large projects, it is now in the position to provide concrete guidance for building software system that meet the architect’s intentions. This guidance can take on the form of informal advices or it is codified in the form of architectural patterns/styles [24]. Architectural styles are proven (functional) software structures that are amenable for certain system properties. The Attribute-Based Architectural Style (ABAS [69]) framework for example, essentially maps quality attribute requirements to suitable architectural styles. As an example consider availability, which is usually defined as up time of a system. For improving availability, redundancy-based styles are suggested including analytical models that motivate that choice.

Software architecture is also about to introduce its own international recommendations for describing software architectures less informal and less ambiguous [61]. In general, describing architectural knowledge and therefore modeling in software architecture is extensively using the notion of views and viewpoints [71]. The central idea behind these terms is to describe the software organization from various perspectives, i.e. in fact describe multiple structures. Each structure captures a set of concerns for a particular group of stakeholders. For instance, software can have:

- The module structure: the structural arrangement of files.
- The call structure of programs and subprograms.

- The process structure: operating system processes and their relationships during the execution of software.
- The physical structure: the allocation of modules or of processes to hardware and therefore also the allocation of inter-process communication to physical communications channels.
- Etc.

Each structure serves for reasoning about a set of related properties. For instance, in the module structure we can reason about what module depends on what others. By this, we know what modules need to be deployed together or what module could be affected by a modification in a certain other module.

While the mapping between views is important it is not usually easy and not properly tool supported because concepts in the various views do not have straight 1:1 correspondences. We believe that the notion of a software component can bring some improvements for view mapping if we restrict components to what Szyperski [131] once called: a module, a process, a unit allocated to hardware, a unit of deployment, etc. Current component-based technology is unfortunately not that restrictive.

2.2.1.1 Architecture Description Language

Specific to the discipline of software architecture are so called Architecture Description Languages (ADL [83]) and ADL-based tools for capturing high-level (i.e. architectural), computational models of a software system. An ADL is used to formally represent an architectural style and its instance in the form of structural arrangements of high-level abstractions: *architectural components* and *connectors*. Architectural components are purely *abstract* computational entities and there is no commitment yet as to how an architectural component is realized as a programming artifact. Hence, architectural components are not a priori software components. Connectors conceptualize specific types of communication protocols among architectural components. An ADL comes with an analytic capability to validate such architectural descriptions, i.e. to predict certain properties (by means of some analytic model). This shall help to analyze early design decisions before a system is realized in the form of software implementations. However, the analytic capabilities are so far limited to a few (e.g. deadlock detection). Up to now, ADLs have hardly reached practicing software engineers. We believe that this is due to the limited benefits compared with the extra effort needed to comprehend a new conceptual framework with its own principles, terminology, and modeling languages.

2.2.2 Component-Based Software Engineering

Component-Based Software Engineering (CBSE) is of particular interest to us because the component-based modeling paradigm can serve as a uniform approach to both systems and software engineering. Further, component-based software is the currently preferred means of developing business-related software. As such, any systems development method or modeling paradigm should make provisions on how to transition to, or incorporate, component-based software as one of the means to realize the software-related parts of a system.

Component-Based Software Engineering (CBSE) can be thought of as a specialization of software engineering with the focus on creating *component software*; this includes both developing software components and, usually at the center of attention, the rapid assembly of components to component-based software systems. The component-based paradigm originally aimed at improving the development economics through systematic software reuse. The (partly still unfulfilled) vision of CBSE is one of assembling software from pre-defined and certified building blocks, which are simply “wired together” in a plug-and-play style. In reality, the wiring still needs a fair amount of so-called glue code.

As the name suggests, component software refers to software that is built from components. While this very general statement is applicable to almost any kind of software (programming language classes are components from which object oriented programs are built, different business applications are components from which an enterprise IT system is built, etc.), component software receives its special connotation by the definition of the term software component.

Software Component. After some debates about what a software component is, it is now commonly accepted to be a piece of software that encapsulates in black-box fashion a *well-defined functionality intended for composition*; i.e. a software component is the unit of composition, is accessible through contractually specified interfaces with explicit context dependencies only, and can be deployed independently [131]. It frequently comes in binary form with pre-defined behavior customization (but the source is immutable!) and it is intended for *third-party* composition. Further, a software component can be put into repositories of reusable components [92]. These definitions would still allow many Commercial-Off-The-Shelf (COTS) software products to qualify as software components. Thus, COTS software products could be glued together in one or the other ad hoc way with the result being called a component-based system. A last and important criterion for a software component is therefore its required conformance to a component model.

Component Model. Because a software component is not realized in isolation but intended for easy composition, i.e. for making it easily interact with other components, component models define some basic rules that components must adhere to. These rules are essentially architectural constraints and usually define at least how components interact with each other. Part of the architectural constraints is frequently expressed in mandatory interfaces that need to be realized by a component and are accessed in a particular way by a component's client. It is the existence of these interfaces and the adherence to well-defined rules that make composition easier for composers (where composers can be thought of as programmers and/or composition tools). Further, it is the same rules that allow component runtime infrastructures to coordinate and support components and their compositions at execution time. This design- and runtime support is part of the reason why a component model is usually three things at the same time: a specification (implying a programming model), a component runtime infrastructure (i.e. some vendor software running on one or more operating system platforms), and a set of general services (like transaction support) on top of the basic runtime infrastructure. At the time of this writing, there are several commercial component technologies, each defining its own component model, in active use: Microsoft's Component Object Model (COM) and its successors COM+ and .NET, Sun's JavaBeans and Enterprise JavaBeans (EJB), and OMG's Corba Component Model (CCM). An introduction to them and their differences can be found in [43].

Interface. Analogous to Simon's observation for adaptive systems [126], the basic assumption of the component-based paradigm is that the relative simplicity of the interface is the primary source of abstraction and generality. Hence, it is the hope that we can characterize a component and its properties without elaborating on the inner workings of the component at all. This fundamental belief leads to the definition of an interface as an abstraction that explicates what a component can do and what a component depends on. While this is in general a powerful idea, current realization of interfaces are focused on decreasing program dependency and on enabling programming language heterogeneity. Hence, they are limited to specifying those properties of a component that a client can programmatically depend on. These are conventionally called functional properties and statically specify a component's services and their signatures (types of services and types and order of parameter arguments of services). The specification of all other properties is a current research issue. Today's interface specifications are usually represented directly in a programming language (like Java in the case of JavaBeans) or in a programming language-like notation (like Microsoft's or OMG's Interface Definition Language, IDL).

In summary, a software component is an architectural entity (by virtue of its component model) and a programming artifact (by virtue of its being a functional black-box ready for integration). But, as can be seen from the definitions, the notion of a software component and as such the whole component-based paradigm is very programming-centric. As with other approaches (like OOP), it

came into being as a way of organizing and representing programs; but not, as a way of modeling complex hierarchical systems. In fact, it is our belief (and documented in our work) that the component-based paradigm can be of much more value if it is taken for what the epistemological meaning of a component stands for, namely to organize complexity by utilization of the part-whole metaphor. Hence, we believe that CBSE shall not only be seen as an engineering discipline built around a programming-paradigm, but as an engineering discipline basing on a powerful system modeling paradigm, which, as a welcomed result, might ultimately lead to software component realizations.

2.3 UML

In this Section we briefly introduce the Unified Modeling Language UML [96]) (with a focus on Version 1.40). While we expect the readers to be familiar with the basic ideas of the de facto standard (including its Object Constraint Language, OCL) and therefore do not present the typical UML basics and supported diagrams at length², we want to emphasize UML's extension mechanisms. They are important in our context for two reasons: (a) we want to represent our own ideas whenever possible in the UML, but the UML does not support some of the very basic requirements we have (e.g. hierarchical system decomposition). (b) The systems engineering community intends the UML to become their future de facto modeling notation too [97] and currently solicits proposals for a systems engineering profile [100]. Our work is a contribution towards the conceptualization of general systems and their properties and as such most appropriate to systems modeling as perceived by the systems engineering community. Consequently, we believe to be able to provide concrete input for such a Request For Proposal (RFP).

As a notational convention we use upper case nouns if we refer to normative concepts defined by the UML specification.

2.3.1 UML Overview

The UML is no more (and no less) than a de facto standard for a graphical model representation language relevant to designing object-based software systems. The size of the UML's alphabet (version 1.40) includes more than 150 symbols, which already partly explains its complexity. The phenomena of interest to be modeled were primarily software programs based on class-based programming language source code (and not systems per se!) but, in the meantime, the UML has been used to represent ever more phenomena of interest. On a spectrum of representations from picture-like to language-like representations [121], UML is clearly a language-like representation. In language-like representations the mapping from a phenomenon to its representation can be entirely arbitrary. The mapping must therefore be established in a persons mind to recognize what the representation stands for. Hence, no effect of the model is achieved without a prior learning of the mapping rules by the interpreter of the model representation. Clearly, standardized agreements on a model language syntax and semantics are needed to transfer knowledge in an unambiguous way either among humans, among computers, or between humans and computers.

The UML up to Version 1.40, which is currently used in practice and supported by CASE tools, cover declarative semantics only. There was one more version (1.50) released but hardly used. It was intended to cover imperative semantics ("action semantics") but it was never really taken up by CASE tool vendors because of the announced major version 2.0, which largely modified and extended also the actions semantics defined in version 1.50. Version 2.0 has been released recently. We will discuss its potential impact on our work in Section 8.2.1.

The UML neither defines a process for creating model artifacts nor what the models have to represent in the domain of interest. Hence, the UML itself is mainly a defined model and its

² We will do so later in the report if a special UML notation is extensively used.

representations to create models: a metamodel framework. However, because a notation always reflects the goals of the architects of the notation (in the case of UML these were three celebrities in the OO field: G. Booch, I. Jacobson, and J. Rumbaugh), UML's semantics of many model elements cannot deny its primary purpose of designing object-oriented software. It is mainly this narrow focus, which leads to variants of UML and the need for newer official versions.

The UML is based on a four-layer metamodelling architecture: meta-metamodel, metamodel, model, and "user objects" (Figure 2-2). For the purpose of future guided extensions to the UML metamodel and for the purpose of model interoperability with Meta-Object Facility (MOF) compliant models, the meta-metamodel of the UML is an instance of the MOF meta-metamodel. However, of immediate practical relevance is the UML metamodel layer, where we find the well-known elements a modeler is using, such as Class, Operation, Attribute, etc. In the model-layer we would find the language elements to describe a particular domain of interest. In the banking domain for example, where we would like to model ATM-related concepts such as CreditCard, showCashOptions, CreditCardBalance, etc.). On the user object layer we would find particular instances of the model elements.

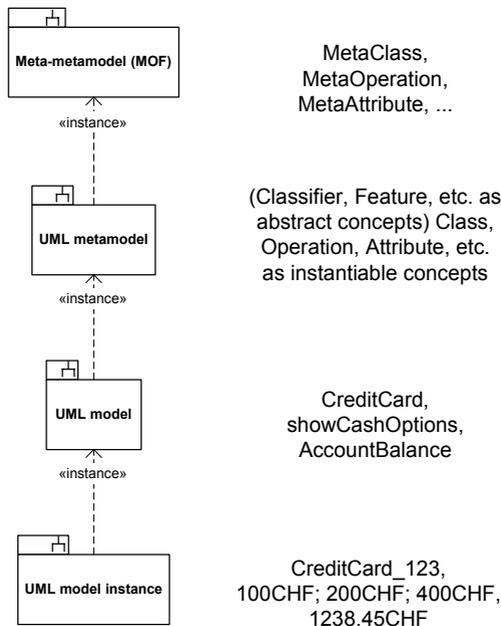


Figure 2-2: UML Meta-Modeling Architecture

In order to position the extension mechanism and also to position the modeling elements most readers are actively using, it is sufficient if we discuss the overall organization of the UML metamodel specification.

The specification of the UML metamodel is divided into the specification of the semantics and the specification of the (graphical) notation. The specification of the semantics is given in natural language supported by conceptual diagrams (or abstract syntax notation as called in the UML), which in turn use UML's class diagram conventions. The semantics specification is organized into three main specification packages as depicted in Figure 2-3: Foundation, Behavioral Elements, Model

Management. The former two are again subdivided into semantically coherent topics of the specification. It is important to mention that the intended Extension Mechanism is part of the Foundation package and as such has a mutual dependency on the Core package of the semantics specification of the UML. Among others, this means that extensions potentially influence many other elements of the specification.

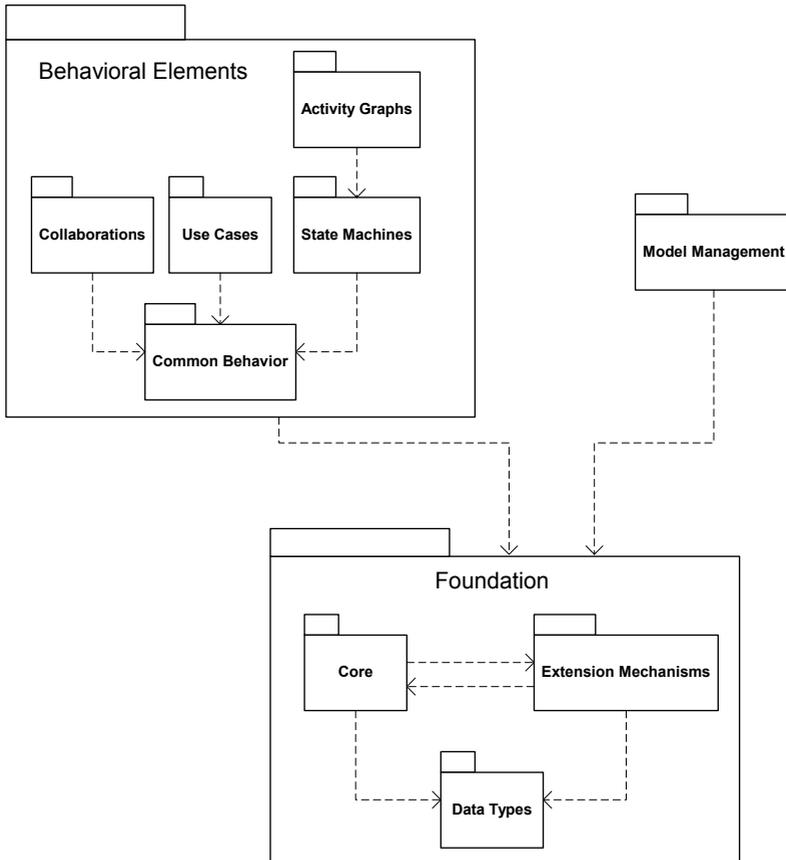


Figure 2-3: Packages of the UML Semantics Specification

The relationships from semantics to the graphical notation are established in that diagrams (graphs) with textual or graphical elements are proposed for the elements of the semantic specification; i.e. representation elements map to semantic elements. The static structure diagrams (class diagram, object diagram) and implementation diagrams (component diagram, deployment diagram) provide notations for elements of the Foundation top-level package and elements of its sub-packages. Analogously, the behavior diagrams (interaction diagram, use case diagram, statechart, activity diagram, etc.) define notations for elements of the Behavioral Elements package.

2.3.2 UML Extension Mechanisms

The UML foresees two principle means of extending its modeling semantics (but really only promotes the first one): a “lightweight” mechanism resulting in UML profiles, and a “heavyweight” extension mechanism based on the MOF compliant addition of new UML metamodel concepts.

As shown in Figure 2-3, the (lightweight) Extension Mechanisms package is part of the semantic Foundation and represents a collection of conventions that specify how UML model elements are customized and extended with new semantics. The built-in extension mechanisms, which facilitate the creation of new modeling elements and the attachment of free-form extensions to any modeling element, rely on the concepts of:

- *Stereotypes*; a model element that refines a specified UML model element through the use of new Constraints, new Tagged Values (based on Tag Definitions), and possibly a new graphical representation.
- *Constraints*; a statement about a model element that must evaluate to true for all the instances of that element. Constraints may be expressed in different ways, preferably in the OCL.
- *Tag Definitions*; a specification of new types of properties that might be attached to model elements.
- *Tagged Values*; the actual set of properties attached to a model element are its tagged values.

A coherent set of extensions, based on the above means, is called a UML *profile*. Several profiles have been proposed and are in use. For instance, the EDOC UML profile [99] is a well-known and exhaustive extension of the UML for enterprise distributed computing.

While any of the mechanisms above could be used in isolation to “mildly” extended the semantics of existing UML model elements, the principal extension mechanism of the UML metamodel is that of a Stereotype. Tag definitions and Constraints are preferably introduced together with a Stereotype because any model element branded by this Stereotype must then obey to these rules. A Stereotype is a means of defining subclasses of UML meta-classes. A stereotype is again a meta-class. A meta-class stands for a class (i.e. a concept) in the UML metamodel specification such as BehavioralFeature (Operation, Method), StructuralFeature (Attribute), or Classifier (the abstract class for instantiable classes such as Class, Use Case, etc). A stereotype specifies its new or extended semantics by any of the following: a natural language text that highlights the specialization over the base UML meta-class, by additional Tagged Values which are defined by Tag Definitions, additional Constraints, and optionally also a new graphical representation. However, none of the extensions may contradict the semantics of existing UML definitions. That is, the extensions leading to a profile are lightweight in the sense that they are refining existing constructs.

Stereotyping is also used by the UML specification itself to define elements that are not considered to be complex enough to justify their existence as an own meta-class. For example, UML’s meta-class called Classifier has defined Stereotypes called <<process>>, <<thread>>, etc. to provide execution-related Classifiers with their extended semantics. Further, a Classifier may have a Tagged Value called “persistence”, which indicates whether the state of an instance of that Classifier is permanent or transitory.

Although not recommended, the UML would allow “heavyweight” extensions. These are definitions of entirely new, i.e. additional, meta-classes or even meta-constructs, which are based on the MOF and not derived from the existing set of UML meta-classes. Heavyweight extensions are cautioned because of the chance to introduce conflicting or redundant semantic definitions, and because of the questionable tool support (e.g. XMI [101] generation).

As we shall see, for general system modeling we suggest both “lightweight” and “heavyweight” extension options.

2.4 Development Methods

Engineering disciplines tend to use the terms method and methodology interchangeably. Because development method rather than development methodology is typically used in systems- and software engineering to denote development approaches, we will use the term method too.³ Our working definition for a systems development method is the following:

A development method refers to a codified conceptual framework that aids the act of applying a systematic approach to create a set of interrelated artifacts, today mostly model artifacts, which ultimately lead to the sought after system that shall exhibit stated (and unstated) desired properties.

The key message of that definition can be presented boldly as in the text frame below.

Development Method = Development Process + Development Artifacts

As mentioned in the previous Section, practicing software or systems engineering is to a large extent a team-based exercise in artifact construction. While artifacts referred to required documents in the document-centric development era, artifacts do now generally refer to models represented in agreed upon model representations. Epistemologically, models are needed to structure knowledge in order to manage complexity and model representations are needed to properly communicate knowledge (see our discussion later in Section 5.1). Collaboration in teams requires such a communication and therefore rules to do so. Development methods codify both a preferred set of model artifacts and their representations and a process with basic rules to facilitate teamwork in creating these artifacts.

Development Process. It is important to mention and it is succinctly described in [29] that today the development process followed by a software project is typically a superimposition of a management process and a technical process. The management process is concerned with planning, progress monitoring, controlling, etc. of the technical process. The technical process is concerned with a “natural” problem solving approach, one that is compatible with our intellectual and cognitive abilities, to proceed from what has to be built to working software (i.e. from requirements over architecture to implementation design and finally executing software). Contrary to the past, today’s technical processes are tuned towards serving the management processes. Evidence for this fact is given by the shift from the early waterfall model (a pure technical process driven development process) to incremental development models such as the spiral model [19] (a heavily management process driven model). Because of its natural appeal, it cannot surprise that we find the waterfall pattern, at least in micro-cycles, in any process invented after it. Software engineering literature documents that we are still having prominent representatives of development approaches that include either or both ([29] p.25): UML Components or Catalysis (both discussed in later Sections) are mainly technical processes, the Dynamic Systems Development Method (DSDM [34]) is primarily a management process, and the Rational Unified Process (RUP [72]) covers both.

Because of our interest in the nature of the model artifacts and in the process of deriving or interrelating model artifacts, we are only concerned with the technical process in our work. Hence,

³ If a distinction needs to be made, we are using the term methodology consistently to represent a “system of methods followed in a particular discipline” [87]. That is, a methodology consists of a set of methods (“a body of methods” [84]) and relationships among them, while a method implies a way of doing something in an orderly logical arrangement of steps [87]. Since in model-based approaches the ordering of activities is of lesser importance, the term methodology would even seem more appropriate.

without explicit mentioning to the contrary we use the term development process without the management process connotation.

Development Artifacts. The development process is intended to provide guidance with respect to a preferred sequence of creating artifacts. With the exception of the concrete system executing at the deployment site, all artifacts can be considered model artifacts. Informally, we define model artifacts as organized descriptions of aspects of systems (e.g. a behavioral model of a software system in execution represented by a UML interaction diagram). The set of model artifacts shall give a good approximation on what properties the “system” will exhibit once it is realized. A development method must therefore be explicit about what the preferred model artifacts are, how they are represented, and how they are related. For example, UML Components (discussed in section 2.4.2.1) defines several model artifacts such as the Business Concept Model, the Use Case Model, the Business Type Model, Interface Specifications, etc. Some artifacts represent orthogonal issues while others can be considered derivatives (e.g., the Business Type Model is a derivative of the Business Concept Model). Fusion [33], a well-known OO method with strict artifact dependencies, defined model artifacts such as the Object Model, the System Object Model, Operation Model, etc. Note that these models are a priori independent of their representation, i.e. they are defined through what aspect of the system they want to express. In the meantime however, many methods have adapted the representations of their model artifacts to the Unified Modeling Language (UML), although they might have started with their own notation (e.g. from Fusion to Fusion/UML [124]). The distinction of models and model representations leads us to revising the bold definition of a development method to the following:

$$\textit{Development Method} = \textit{Development Process} + \{ \textit{Development Artifact}, \textit{Development Artifact Representations} \}$$

It is important to mention that the central model artifacts of a software development method circled around the representation of the software to be built from various perspectives and not the system to be built. In fact, all the major modeling paradigms (structured analysis and design, object-orientation) were born as program abstractions rather than modeling world-views. They were then adopted and further extended to suit modeling of artifacts that are not abstractions of program code or program execution. Examples are the introduction of use cases, or requirements modeling in general. This trend simply acknowledges the importance of the environment of the software and the dependency that creating the “right” software has on understanding the world around the software programs. As a consequence, program-centric models are increasingly augmented with models capturing the software’s environment (hardware, business, etc.). This is mainly to make sure and communicate that the software fulfills the part it is intended to fulfill. Even for this enlarged scope, the prevailing model representation language of choice is the Unified Modeling Language (UML), although it cannot deny its original focus on modeling object-oriented software.

In conclusion, the currently produced model-based development artifacts suffer from two potential shortcomings: the models were intended to capture software related phenomena and the prevailing model representation concepts were specifically defined to represent software concepts. If software engineering is promoted to systems engineering, we feel that this is justification to also consider other approaches or semantic redefinitions/extensions of the software related approaches for general systems modeling. Given the different types of software-intensive systems to be realized and thus the different types of model-based development artifacts, together with the different types of management processes, it cannot surprise that there is a need for method engineering as expressed in Section 1.1.3. The visible result is a proliferation of development methods that are created mainly by large software developing companies and consultants.

2.4.1 Systems Development Methods

Analogous to software engineering, systems engineering is currently transitioning from a document-centric era to a model-driven one. But systems engineering lags somewhat behind. The

process of creating envisioned systems is largely still organized around creating artifacts in the form of documents, but it is now more often (and shall in the future be) organized around creating artifacts in the form of models (i.e. model representations).

As representatives for model-driven approaches, we briefly present two systems development methods: OOSEM and OPM. While OOSEM is promoted by the systems engineering community, OPM is hardly known. We include it here because of its strong emphasis on an underlying systems foundation.

OOSEM (Object-Oriented Systems Engineering Method [76]). OOSEM is the most well known method and, because of its partial UML usage, also assessed for its suitability to serve as a system engineering modeling standard (discussed in Section 2.5 below). OOSEM has evolved from a process framework for the integration of systems- and software engineering methods to a process framework that partly proposes its own modeling method based on the UML. Modeling in OOSEM refers to creating model representations that capture the system to be developed. OOSEM's emphasis is on capturing system requirements in a model-based fashion, and in facilitating the breakdown of these requirements and allocation to system components (hardware, software, people, manual procedures). OOSEM defines *the* system as the software-intensive system to be developed. While OOSEM proposes model artifacts of its own, it is open to have its proposed artifacts replaced or supplemented by specialists' models.

The main OOSEM activities and their modeling means are compiled in Table 2. Details on new terms used in the "Model artifacts" column are described after the table.

Table 2: OOSEM Activities and Main Model Artifacts

<i>Activity</i>	<i>Model artifacts</i>
Analyze needs	"as-is" and "to-be" use cases, scenario descriptions, static model of the system and its collaborating external systems and human actors. UML Use Case, Sequence Diagram, and Class notations are used.
Define system requirements	Elaborated system context diagram: a class diagram depicting the system as a black-box unit, its I/O entities, and its external collaborators.
Define logical architecture	Composition hierarchy of <i>logical</i> components depicted as a UML Class aggregation hierarchy
Synthesize candidate allocated architectures	The allocation of <i>realization</i> components to the logical components may use any UML modeling means that suits the purpose
Optimize and evaluate alternatives	The notion of parametric models is used. It refers to logical components and concrete components whose property values (represented as UML attributes) may be parametric.
Validate and verify the system	None in particular

OOSEM promotes requirements traceability and a model-based approach, but it does not accomplish the traceability in a model-integrated way. It traces requirements (and also captures other constraints) external to the models by means of a so-called Requirements and Verification Traceability database (RVT). The latter has to be maintained by the systems engineers.

What is specific about OOSEM's modeling approach and model representations?

- It essentially uses stereotyping of UML modeling elements. The most prominent concept is that of a system, which is represented as a UML Class. More over, almost all

introduced system's concepts are represented as UML Classes (I/O, logical components, etc.).

- The system, as a black box, is modeled by (a) its services (called functional behavior in OOSEM and represented by capabilities, which is a stereotype <<cap>> of the concept of a UML Operation) and by (b) properties that relate to system state, transient and persistent data, and performance characteristics, such as timing, size, etc. All properties besides services are modeled by UML Attributes and grouped into a few categories, each of which is defined as stereotype of a UML Attribute.
- OOSEM uses hierarchical system decomposition into so-called Logical Components. OOSEM calls them an abstraction of a set of possible system components, i.e. they are logical containers that hold a specification. Any realization would in principle do, if the specification were met. For example, a user-interface could be a logical component, whose realization might be a Web-browser on a PC or a dedicated user console.
- OOSEM makes input/output entities (I/O) explicit and represents them as stereotyped UML Classes. Class Attributes of I/O entities can be "physical" (stereotype <<phys>> and representing what OOSEM calls mass, energy, or parts) or "data" (stereotype <<data>> and representing information contents transferred by means of data communications).

OOSEM provides both a suggestion for a development process and for development model artifacts and their representations. It proposes a hierarchical decomposition that is similar to the ideas of ABD (discussed in Section 2.4.2.2), namely recursive decomposition into logical components (or conceptual components as called in ABD). However, no guidance is given on how the decomposition shall be accomplished. For almost all model representations, OOSEM suggests stereotyped UML Class diagrams with stereotyped UML Operations or UML Attributes, even if the usage is partly inconsistent with the UML semantic definition of these concepts. The choice of models, however, is not made explicit and consequently inter-model dependencies weak. Further, there does not seem to be a rule or a philosophical underpinning that defines the types of supported properties. For instance, OOSEM defines so-called *effectiveness measures* (e.g. "wait time" for clients in front of an ATM), which is one type of system-level property and OOSEM defines so-called *system performance parameters* (e.g. "withdraw time", or "power consumption") as another type of property. However, the distinction between the two types and how they are used, aside from being represented as stereotyped attributes, remains unclear. As mentioned earlier, the traceability among model artifacts, in fact among any documented piece of information pertaining to the development of the system, must be accomplished external to the models. This introduces a huge burden for the developers in terms of keeping consistency among models and between models and the RVT.

OOSEM, as a dedicated systems engineering method, shows a clear sign however, for the need of incorporating not only information but also matter/energy flow, non-functional properties, and system hierarchies.

Object-Process Methodology (OPM [37]). OPM is an integrated modeling method that unifies the system structure and behavior within one frame of reference. OPM is *not* a development process framework. OPM claims both a human factors basis and an ontological systems basis.

The human factors basis can be expressed in saying that OPM's premise is that structure and behavior modeling shall not be segregated into separate diagrams and notations, as is the case in the UML, because human comprehension must then reunite these separate views to a consistent mental model. Hence, OPM does not use the UML but instead has its own visual and textual representation.

Further, OPM claims a system-theoretical ontological foundation because it is not primarily derived from OO programming concepts (i.e. bottom-up) but instead takes a plausible foundation that

“views systems as composed of objects transformed by processes, which generate, consume, or affect objects by changing their state”[38]. We classified OPM as a systems development method, although in literature OPM applications are mainly found in the context of the development of software. We did so, because of its foundational, domain-independent systems view. This basic stance has also influenced the rather lean set of basic modeling building blocks: *objects*, *processes*, *states*, and structural and behavioral *links*. Each of these basic building blocks has defined semantics and some have defined subtypes. Most importantly, OPM is the only method, which we know of, that cleanly distinguishes between physical and conceptual objects and correspondingly physical and conceptual processes.

Every OPM system description (or specification as called in OPM) is composed of a set of graphical and textual, i.e. natural language, representations, which correspond to each other. The graphical representations are called Object-Process Diagrams (OPD), the textual representation Object-Process Language (OPL) scripts. To give an intuition for these representations we have included excerpts of Figure 2-4 from [105]. Translation between both kinds of representations is done automatically.

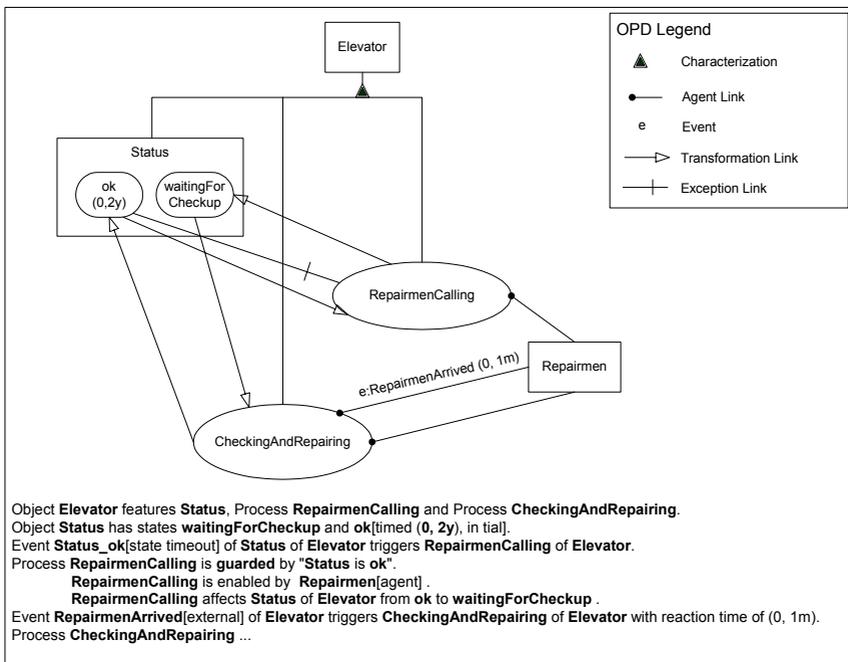


Figure 2-4: An OPD and Equivalent OPL Text of an Elevator Checkup Case Study

Figure 2-4 represents the specification for a system requirement that was formulated as follows:

“The elevator of a building has to be checked every two years. When the time for a checkup of the elevator arrives, repairmen are summoned. Within one minute of the repairmen’s arrival to check the elevator, the elevator is checked, and if necessary, repaired, so that it is again in a usable condition.” [105]

Without going into every detail, we explain the figure briefly. The top of the OPD, as well as the first OPL sentence, specify the features that characterize the elevator: its **object status**, and its two characterizing **processes** (methods, or services). The object status is defined by two states, one of which is constrained by a duration (“ok” by (min, max) time to stay in that state). After that time a time out event triggers the RepairmenCalling process. This event is shown as the exception link. The process then changes the state to waitingForCheckup (indicated by directed arrow). Note, as mentioned in the system-theoretical foundation in the beginning of this Section, states can only be changed by processes. This short sketch of an OPD and its OPL should be sufficient to give a flavor of OPM’s model representation approach, for instance as opposed to the UML.

So what is missing for systems engineering? OPM supports hierarchies through nesting of objects. But OPM primarily deals with behavior and structure modeling for functional requirements. The only non-functional property truly supported is timing constraints (by the so-called OPM/T extension). Provisions for model-based inclusion of any other properties are not made. Further, input/output and in general information/matter/energy flow (often required by system engineers) is not explicitly modeled.

2.4.1.1 Other Systems Development Methods

There are other systems development methods or processes known in systems engineering (e.g. ISSEP [122]). However, like standards such as the ANSI/EIA 632 “Processes for Engineering a System” [42], they primarily address the systems engineering development *process* but not the individual methods and model artifacts to be used and produced, respectively. In other words, although they model the complex process of executing systems development projects, they are not model-centric in the sense that they are not specifically proposing an approach to modeling the system “to-be-built” or intermediate model artifacts. They rather provide a framework that focuses on the related activities and informally describe the suggested artifacts to be considered while running a systems development project.

2.4.2 Software Development Methods

Software development methods have progressed from structured- over object-oriented- to component-based approaches. The latter build upon their predecessors in that they combine and/or modify the methodological concepts behind the dominant object-oriented methods to suit the specifics of component technology. It is therefore sufficient to briefly review the major component-based development methods to document the state of the art in software development methods. To this end, we present Catalysis, UML Components, and Kobra to emphasize their approaches to modeling (hierarchical) systems and to modeling various types of properties. In addition, and because of the complementary nature of software architecture to CBSE, we discuss a software architecture-driven method (the ABD).

2.4.2.1 Component-Based Software Development Methods

Catalysis [40]. Catalysis was the first component-based development method widely accepted, at least in academia. Similar to Kobra (discussed below), Catalysis builds heavily on the Fusion method [33], which was designed to develop OO software. Catalysis emphasizes work products, i.e. the model artifacts, more than a strictly prescribed process.

The Catalysis world-view is that of collaborating objects. Consequently, its central idea is that of rigorously specifying collaborations of objects. Objects are identifiable components with an interface. As a consequence, the central modeling paradigm revolves around refinement of collaborations and refinement of components. A collaboration is a set of transactions among components with common purpose and a common level of abstraction.

Catalysis’ underlying premise is to pursue a “what-who-how” refinement process. “What” refers to describing joined transactions, i.e. the effect of the joint behavior of collaborating components.

“Who” refers to defining localized transactions and to designing the interactions that yield the joint effect. Localized transactions are assigned responsibilities to the individual participants of such a collaboration and as such represent the services of a certain component. Finally, “How” refers to the refinement step in which components are refined into finer-grained components and in which interactions are refined into finer-grained interactions. Similarly to Fusion’s notion of an operation schema, transactions are generally formalized by means of predicate logic, i.e. they relate pre- and after transaction states with the help of a pre- and post-conditions formalism.

Originally, Catalysis defined its own graphical notation based on Rumbaugh’s Object Modeling Technique (OMT [119]). Over time, users have adopted the concepts and principles of Catalysis but tried to apply UML as the means to graphically represent model artifacts. From a UML point of view this is doable but requires extensions that are hardly supported by current UML CASE tools.

In summary, Catalysis would in principle be able to deal with hierarchical systems because it follows a recursive component and collaboration decomposition process. However, it was not intended to do so and it defines three specific levels based on the Fusion principle: problem domain or business modeling (“outside view”), component specification (“boundary view”), and component implementation (“inside view”). Catalysis makes no provisions to explicitly deal with non-functional properties of any kind. Model-based traceability is supported in principle but limited to tracing component or transaction refinement relationships.

UML Components [29]. UML Components is an excellent, because practical, approach to the design of software systems based on software components. It has a rather pragmatic focus on specifying application software components and for this purpose defines its own extensions to UML. The basic premise of UML Components is to build IT-based information systems with a classical three-tier architecture and concentrates on the specification of components for the middle tier, the business logic. Besides its compelling brevity, the major contributions of UML Components are the justified UML extensions and the notion and usage of interface data models and techniques to specify which component interface is responsible to manage which data item. Interface data models are data models that are relevant for individual component interfaces only.

Although UML Components is a model-based development method, it still defines a prescriptive development process, as a set of workflows. And it defines specific model-based artifacts as required workflow in- and output. Figure 2-5 depicts the four main workflows and its activities – requirement workflow, specification workflow, provisioning workflow, and assembly workflow – and it lists the relevant model artifacts per workflow. As can be guessed from the figure, UML Components is strong in its emphasis on the requirements and specification workflows, where it suggests a concise way by means of activities, which guide from use case based system requirements to software component specifications through a progression of defined model artifacts.

Opposed to Kobra for instance, UML Components does not claim a component-based modeling worldview for hierarchical systems. Hence, it does not employ a recursive approach to component modeling although it mentions the concept of subcomponents, but only as a programming convenience. Subcomponents and components are not treated the same. Further, besides its lack of addressing user interface issues, UML Components falls short in explicitly dealing with non-functional properties of any kind. Because use cases are the key starting point, it addresses end-user oriented functional properties only.

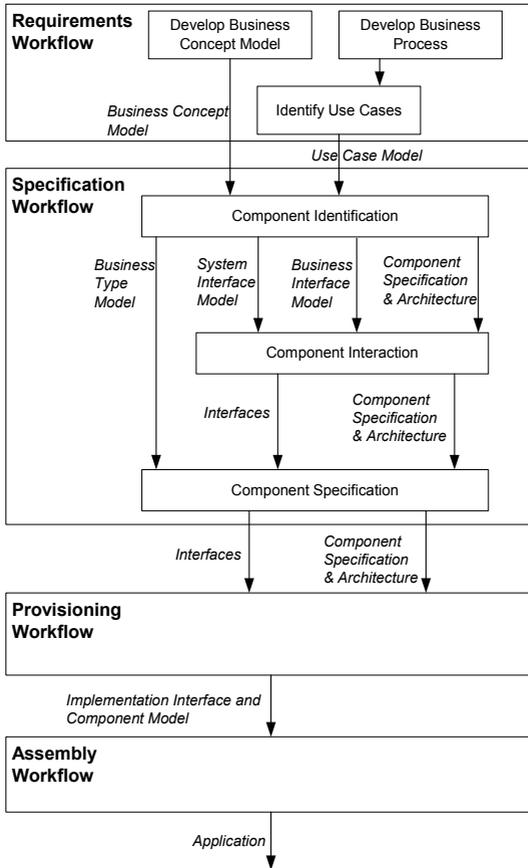


Figure 2-5: Workflow and Model Artifacts of UML Components

KobrA [4]. The KobrA method defines a fractal-like model with a recursive component composition hierarchy as the main vehicle to realize a uniform modeling approach on hierarchical system levels. It is model-based and uses UML as the model language of choice. All modeling activities are component-centric. The fundamental metaphor of KobrA is: *component = (sub)system*. Every component is a system (subsystem) and "...any behavior-rich software abstraction capable of remembering some state information and/or responding intelligently to external requests can be a component..."[5]. KobrA introduces so-called specification and realization models that closely mirror Fusion's [33] ideas of analysis and design models, respectively. Accordingly, they correspond to the outside- and inside views of a system. The specification model is used to describe the externally visible properties of a component and, like Fusion, is composed of three model views: a structural view using static structure diagrams, a functional view expressed in operation specifications represented in predicate logic, and a behavioral view represented by state diagrams. The realization model describes the internals of a component, i.e. how it realizes the external visible properties. It also

uses three main model views: a structural view using static structure diagrams, an interaction view that describes how the component interacts with its environment and its internal components by means of interaction diagrams (UML collaboration or sequence diagrams), and an algorithmic view that uses activity diagrams to show how a service, most notably an operation, is realized.

Figure 2-6 shows the Kobra's canonical picture to illustrate their use of the aforementioned models.

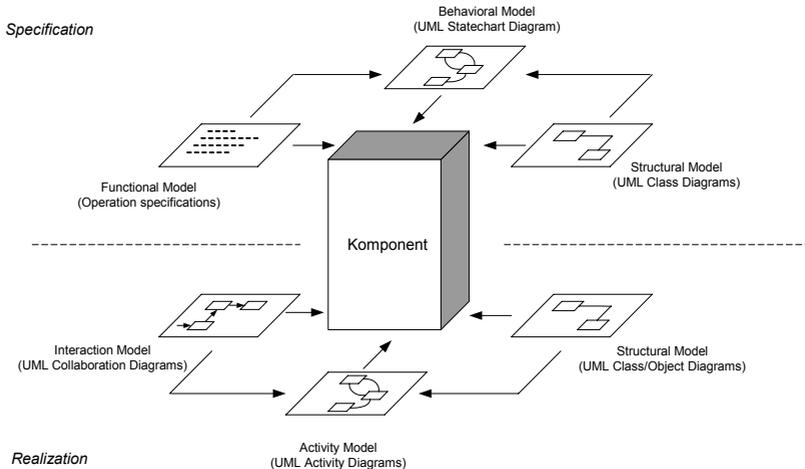


Figure 2-6: Kobra Specification and Realization Models (Source [5])

As can be inferred from the type of models used, Kobra supports a model-based approach to the design for functional properties of software systems. Non-functional, execution-related properties are not addressed. Some non-functional, development process related properties, are addressed through a minimal quality assurance mechanism, which is part of the suggested development process [4].

In summary, Kobra suggests a hierarchy of systems where the same modeling approach is applied on every system level. However, the notion of system or component is restricted to *software* abstraction. In fact, Kobra's main difference to Fusion or OMT is that the discussed modeling approach is applied recursively, which results in a hierarchy of nested components. Kobra makes no difference between conceptual and concrete components. It is only interested in recursive decomposition to be able to create one type of behavior template: that of software. Hence, it does not claim to model general systems. Kobra uses the UML as the language to represent its defined models but, in order to accommodate for their definition of component, has to "bend" the UML semantics. The Kobra modeling approach does not support non-functional properties.

2.4.2.2 Architecture-Based Design Method

In this subsection we briefly introduce a method to specifying software architectures: ABD, Architecture Based Design [9]. Although ABD is not a full-fledged development method, we include it in our review, as an example of the rare disciplined approaches explicitly addressing a wider range of non-functional properties.

ABD is not a comprehensive development method but an approach that emphasizes the architectural design phase of a *software* development. Hence, a software development method could use ABD as an integral part aimed at focusing the activities of the architect in drafting the basic high-

level architecture. In the development life-cycle, ABD would follow the requirements engineering phase and end before the detailed software design, i.e. end before there are any commitments to programming language artifacts. The assumed inputs for ABD are so-called architectural drivers, which are divided into functional requirements, quality requirements of different kinds, and business requirements. As an example for such requirements consider an Automatic Teller Machine (ATM). A functional requirement is that “it shall dispense the requested amount of money if guaranteed through the identified bank card”. A quality requirement of type modifiability is that “the system (the ATM) shall be modifiable with respect to the type of currency that is used”. Finally, a business requirement may be that the ATM development company has invested in a product line business and requires that “the display and display drivers be reused”. The fuzzy delineation between quality and business requirements, however, causes ABD to practically lump them together into one type of input. The outputs of ABD are so-called conceptual components, i.e. architectural components in the form of behavioral units that are part of interaction patterns. ABD does not define how these conceptual components have to be realized.

The foundation of the ABD approach rests upon recursive functional decomposition as a means to cope with functional requirements and rests upon architectural styles as a means to cope with quality and business requirements. The basic philosophy is to decompose the application into components and their responsibilities (the functional decomposition) and to merge such a decomposition with an infrastructure, which is an instance of an architectural style existing of “empty” architectural components (called templates in ABD) and interaction patterns. By this merger the application components fill the empty architectural components and also inherit the needed responsibilities to adhere to the needed interaction pattern. Any decomposition is modeled and then analyzed based on three views: logical view, concurrency view, and deployment view. The logical view captures the conceptual components and their roles in the form of their interfaces. The concurrency view captures issues such as multiple users, parallel processing, resource contention, or component startup behavior. The deployment view captures the physical structure of the system in terms of processing nodes and communication links. The analysis takes on the form of examining functional requirements by conceptually executing use cases and of examining quality and business requirements by conceptually executing quality scenarios on the derived design.

ABD is a recent example for design guidance that can take on the form of codified, yet informal advices. ABD is not explicit as to the nature of properties it can and cannot cope with. It is still too young to comment on it based on documented experience. ABD does not define a modeling language. The ABD approach foresees recursive decomposition of systems. However, what is meant is a recursive decomposition of conceptual components, if needed. Hence, it states that two levels of decomposition are usually sufficient: system decomposed into subsystems, which in turn are decomposed into conceptual components. ABD is not explicit on how traceability between decomposition levels or how consistency between views has to be established.

2.5 An Envisioned Systems Engineering Modeling Standard

As mentioned above, systems development methods are becoming model-centric and systems engineering is one of the disciplines where different specialty areas have to collaborate closely to create viable systems solutions to complex problems. Hence, communication among specialists across disciplines is a prime need in systems engineering. This communication shall be supported better through an agreed upon basic terminology and by models with agreed upon model representations. A central issue is the definition of a system and its model representation. The systems engineering community is therefore currently very active in conceiving a “concept model for systems engineering” [94] as well as evaluating the adoption of the UML for systems modeling [98, 100]. Both activities have started recently and our work aims to provide valuable input to them. The concept model refers to an agreement on a unified semantic dictionary of general terms used in systems engineering, which shall be as generic as possible, i.e. applicable to the various systems engineering domains. Central to

this concept model is the notion of a system and its properties. The adoption of UML requires a careful evaluation, selection, and extension of UML modeling elements to suit the purpose of general system modeling. The UML is mainly chosen because software engineering is one specialty area that is called upon in almost all of today's systems development efforts (recall Section 2.1 and particularly Table 1) and the UML is the de facto model representation standard in software engineering.

The means of the OMG to arrive at a concept model and also to arrive at a UML profile for systems engineering is essentially an inductive one. That is, various parties are requested to provide their inputs and requirements on concepts they are using in their specific domain and on possible UML modeling elements to represent them. We believe that this results in a huge effort because it leaves the receiving and coordinating OMG committee with the job to weave and potentially abstract a large number of concepts, described at various levels of details, with a heterogeneous terminology.

We prefer a deductive approach in which we start with a small theory, i.e. with a small set of systems concepts and see whether they can be taken as generalizations of the concepts of the specialty fields. If so, domain-specific aliases or specializations can serve for the domain customization. And domain specific requirements can be tested against such a first, general conceptualization. This is one more reason why we wanted to develop a holistic model of systems and properties.

The central requirements on a standard systems modeling language can be summarized as follows:

- a) It must be possible to model concrete systems (e.g. car) as well as conceptual systems (e.g. software).
- b) As a consequence of a), it must be possible to model matter, energy, and information processing.
- c) It must be possible to model hierarchies of systems, i.e. hierarchical system decompositions and not only support view- or aspect-based decompositions (see paragraph immediately following this bullet list). This includes the support for modeling of system environment.
- d) Besides expressing the structure and behavior of a system, it must be possible to express different types of other properties such as physical properties, performance, effectiveness, etc.
- e) It must be possible to trace system and property realizations (or sometimes called property decompositions).
- f) It must be possible to express physical structure, i.e. concrete components, system boundary and dedicated system access points as well as connections of systems/components.
- g) Flow of information as well as matter/energy (e.g. physical entities such as parcels) must be explicit. This includes the capability to distinguish between consumed input (resources) and "copied" input.
- h) The modeling language must be open to extensions/specializations needed to accommodate for engineering domain specific concepts.
- i) It shall be based on UML, i.e. be compliant with the UML metamodel.

Concerning hierarchical decompositions as mentioned in c), we should mention that current OO modeling methods typically approach decomposition in other ways. They do not really view the universe of discourse to be modeled as a hierarchy of systems and consequently decompose the model of the perceived reality into hierarchical levels. They rather decompose a system description into intellectually manageable pieces. The set of pieces takes on the form of a collection of models, where each model provides a partial view of the system. Further, many decomposition-like approaches

decompose a system into logical rather than concrete pieces. These logical pieces, for instance behavioral abstractions like architectural components, are not concrete systems that will be deployed into reality as self-standing entities. They rather serve as intermediate constructs to facilitate human reasoning.

The requirements of major concern to us are a) to e) and i). If we compare the modeling approaches of the aforementioned development methods in software- and systems engineering against these requirements, we obtain the situation as listed in Table 3.

Table 3: Comparison of System Engineering Relevant Features

	Catalysis	UML Components	KobrA	ABD	OOSEM	OPM
a) Concrete (K) and Conceptual (C) systems	K – No C – yes	K – No C – yes	K – No C – yes	K – No C – yes	Yes	Yes
b) Matter/energy (M) and Information (I) Processing	M – No I - Yes	M – No I - Yes	M – No I - Yes	M – No I - Yes	Yes	M – No I - Yes
c) Hierarchical Concrete (K) and Conceptual (C) Systems	Partly	No	Partly	K- No C - Yes	Yes	Yes
d) Functional (F) and Non-Functional (NF) Properties	F – Yes, NF - No	F – Yes, NF - No	F – Yes NF – very limited	Partly	F - Yes NF partly	F – Yes, NF - No
e) Model-integrated Property Traceability	Partly	Partly	Partly	No	No	No
i) UML- based	No (but could be)	Yes	Yes	No	Partly	No

Based on this table we can see that there is no method currently available that attempts to accommodate in a model-based and traceable way for (i) a wider range of properties, i.e. non-functional properties, *and* (ii) hierarchies of concrete and/or conceptual systems, *and* (iii) that attempts to represent both – (i) and (ii) – in the UML.

In order to eventually suggest how to extend or modify existing development methods, or suggest a new method if really needed, we wanted to investigate and set forth the required theoretical preliminaries in this thesis. By preliminaries we mean the methodological building blocks that result from a more fundamental understanding of the different nature of properties of systems.

3 The Different Flavors of Properties of Systems

Objectives: As part of the context of this work, this Chapter presents the current understanding of the different types of properties to be dealt with in systems- and in software engineering. To reflect the current state and its inconsistent terminology the term property inevitably covers synonymously used terms such as attributes, qualities, quality attributes.

After having read this chapter, the reader will:

- have developed an intuition about the kinds and classification of properties used in systems- and software engineering;
- know the conceptual approach and the limitations of decompositional quality models such as the ISO/IEC 9126, which essentially try to suggest taxonomies of quality attributes;
- know that in software engineering the term quality, as opposed to the general linguistic understanding, has the special connotation of being related to a stated or unstated desire;
- realize that a common theory on properties of systems is, at best, a fragmented one and a conceptual cleansing would be appreciated.

Because dealing with an object's "as-is", "to-be-built", or "as-built" properties is key to every engineering discipline, there is an abundant number of types of properties and consequently there is and must be an abundant number of type-specific definitions. These abundances stem from the fact that properties are related to the types of entities we ascribe them to, and they are related also to the purpose of the observers or stakeholders that are taken into consideration. However, when it comes to generalizing or reducing the wide range of different types of properties to those that have an intrinsically different nature, there is fewer works available. Frequently, it is some form of standard that attempts to conceptualize and classify a wider scope of property types. Such standards are the result of earlier research work combined with the gained practical engineering experience by the many working group members for such a standard. For example, the ISO/IEC 9126, discussed later, is the result of works by McCall [80] and Boehm [21]) and subsequently gained experience with them.

Most of the current research, however, focuses on property-type specific theories and methods. For instance, there are general theories on dependability or reliability [135] but also specific theories on software reliability [52] [104]. All these specific theories and methods have in common that their property models are hardly generalizable. Some works is taking a more holistic approach to defining property (or quality) models *but* they are then usually constrained to a specific entity. For example, Bertoa et al. [17, 79] or Shaw [125] concentrate on defining a property model for software components. Dromey [39], Bansiya [11] or Puraio et al. [113] focus on metrics for OO design concepts such as class, method, module, etc.

In the following two Sections we will reflect the current holistic understanding of properties of systems or of products mainly based on the most prominent and also frequently cited standards. We will address the models of the entity-specific properties, as defined by other researchers, in the topically relevant Sections of this thesis report.

On order to not get confused in the subsequent Sections, Figure 3-1 shows an informal illustration in which the referenced standards or basic literature are positioned.

The property classifications of the standard “Processes for Engineering a System” (ANIS/EIA 632) and of the “Guide for Developing System Requirements Specifications” (IEEE Std. 1233) cover the wide range of systems as found in systems engineering and could potentially be applied to systems that do not even include software products. The object for which these two standards categorize properties or requirements is the system seen as a black box. Sommerville’s classification of system requirements refers to computer-based systems. That is, it also addresses the system as a black box, but with clear assumption that major parts of the system are software products. The “Recommended Practice for Software Requirements Specifications” (IEEE Std. 830) as well as the “Software engineering - Product quality - Part1: Quality model” (ISO/IEC 9126-1) clearly focus on the software product and secondly on those aspects of the system that are closely related to the software product, e.g. when the software product is in execution in its deployment environment.

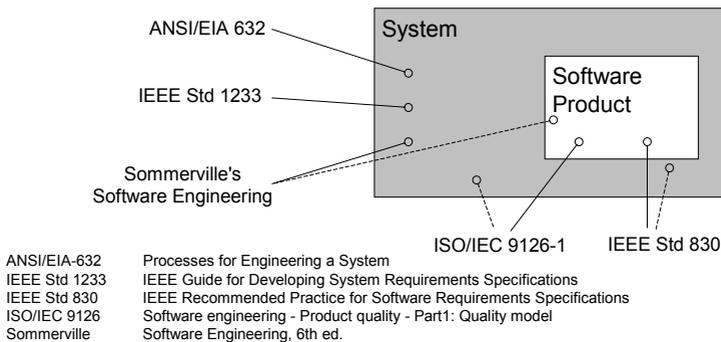


Figure 3-1: Positioning of Standards

A remark on requirements engineering

In general, any systems development undertaking starts with the identification or collection of requirements on the envisioned system. These requirements come from various sources, typically related to different needs expressed by different stakeholders. After the collection of the various requirements is accomplished, they need to be analyzed for their feasibility and for their conflicts. A process of requirements negotiation is then undergone to resolve infeasible requirements and conflicts. Both requirements collection and requirements negotiation are fields of study on their own and subsumed under the term requirements engineering.

Requirements engineering in general, and negotiation in particular, is not within the focus of our research work and consequently not within the focus of this thesis report. It is our assumption that when we speak about the set of required properties or “to-be-built” properties of a system, they have gone through and are the result of a requirements negotiation process. Hence, the conflicts are resolved and the required properties are the assumingly conflict-less, prioritized properties. For example, with respect to non-functional execution-related properties, say usability and security, this would mean that the negotiation has resulted in the fact that security is top priority and its realization may compromise usability. Or, with respect to non-functional development-related properties, changeability of functionality is more important than installability of the software.

3.1 The Systems Engineering Viewpoint

Common to all holistic models of properties or requirements in systems engineering is that these models are reduced to a classification of property or requirements types. They do not define an explicit conceptual model or meta-model of generic terms like for instance the ISO/IEC 9126 does. We therefore defer a discussion of the notion of a *quality model* until Section 3.2.1.

3.1.1 Properties and Classifications of Requirements in Systems Engineering

3.1.1.1 ANSI/EIA 632

The ANSI/EIA 632 standard “Processes for Engineering a System” [42] does not explicitly introduce the term non-functional property or non-functional requirement, but enumerates different types of requirements. This enumeration starts with functional requirements and then performance requirements, interface requirements, etc. Hence, they also adopt a partitioning into functional requirements and “others”. The ANSI/EIA 632 distinction of requirements has been taken over by the OMG special interest group for systems engineering in their currently still evolving “concept model for systems engineering” [94]. In Figure 3-2, we show the part of the concept model that defines a classification of these requirements. We have put the informal definition as used in the concept model into UML notes. It should become evident that the classification with the given definition is clearly a pragmatic one, mainly to provide guidance on what requirements to look for when compiling requirements documents. It is not particularly helpful to find fundamental commonalities and differences for the basic types of properties that are related to the basic types of requirements. Thus, the task of deriving general property models or deriving patterns to include properties in a uniform way into models of systems is not facilitated by this kind of classification.

It is interesting to mention that the definition for “Reference Requirement” (which we shortened a bit in Figure 3-2) lists basically all so-called Quality Characteristics as defined by the ISO/IEC 9126 [63]. The latter is discussed in Section 3.2.1.1 below as a *software* engineering related model. This is interesting insofar, as that it amounts to the fact that the set of relevant properties in systems engineering is a superset of properties of the widely accepted properties for software products defined in ISO/IEC 9126. The superset in this case consists of those properties that are covered in Figure 3-2 under Interface-, Physical Property-, Imposed Design Requirement, and Effectiveness Measure. Sommerville’s frequently cited non-functional requirement classification in software engineering ([128] p.100ff and Figure 3-3) is another example that supports this point, in that his classification could also be used to further divide the “Reference Requirement” type. Not surprisingly, these systems engineering specific properties reflect the fact that systems, which are dealt with in systems engineering, process not only information but also matter and energy and therefore consider a larger number of system types.

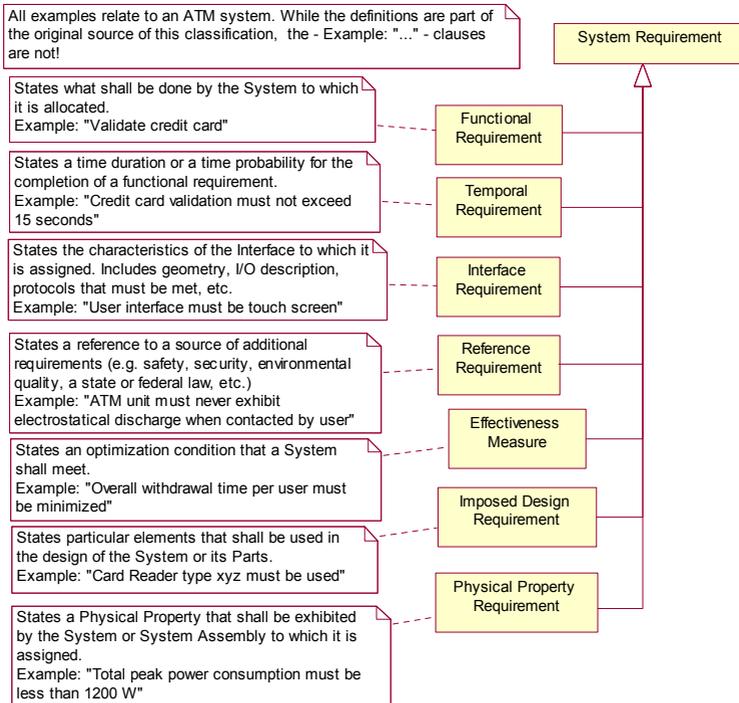


Figure 3-2: Requirement Types as Defined by the ANSI/EIA 632 (Adapted from [94])

3.1.1.2 IEEE Std. 1233

Similar to the systems engineering standard ANSI/EIA 632, the notion of non-functional property or non-functional requirement is not used explicitly in the IEEE 1233 standard “Guide for Developing System Requirements Specifications” [58]. The choice for this categorization, i.e. the criteria for categorization, and the definition of the base categories of requirements types is less intuitive. To prove our statement, the contents and wording in Table 4 is taken directly from the IEEE Std. 1233. As can be seen, it assumes a flat hierarchy and it also assumes that the names of the types are self-explanatory. The IEEE Std. 1233 only provides definitions by (one) example as captured in the Definition column of Table 4. All but the Input and Output requirement types could potentially be grouped according to the main types of the ANSI/EIA 632 (recall Figure 3-2), mainly because of its flexible *reference requirement* type. Another way of grouping the IEEE Std. 1233 requirement types would be to accommodate them under one of the non-functional requirement types defined by Sommerville [128], discussed next and depicted in Figure 3-3.

Table 4: Requirement Types as Defined by the IEEE Std. 1233

Requirement types	Definition as per IEEE Std. 1233
Input	e.g., receive EDI data
Output	e.g., export a particular format
Reliability	e.g., mean time to failure
Availability	e.g., expected hours of operation
Maintainability	e.g., ease with which components can be replaced
Performance	e.g., response time
Accessibility	e.g., different navigation paths for novice and experienced users
Environmental conditions	e.g., dust levels that must be tolerated
Ergonomic	e.g., use of particular colors to reduce eye strain
Safety	e.g., below specified limits for electrical magnetic radiation
Security	e.g., limits to physical, functional, or data access, by authorized and unauthorized users
Facility requirements	e.g., use of domestic electrical current
Transportability	e.g., weight limits for portability
Training	e.g., includes tutorials or computer-based training
Documentation	e.g., on-line help facility
External Interfaces	e.g., support for industry standard communication mode/format
Testing	e.g., support for remote diagnostic
Quality provisions	e.g., minimum required calibration intervals
Policy and regulatory	e.g., environmental protection agency policies
Compatibility to existing systems	e.g., uses analog telephone system as default mode
Standards and technical policies	e.g., products to conform to ASME codes
Conversion	e.g., will accept data produced by older versions of the system
Growth capacity	e.g., will support an additional number of users
Installation	e.g., ability to put a new system into service

3.1.1.3 Requirement Types Defined in “Software Engineering” (Sommerville [128])

As a last representative for a classification of requirement types, we present the one suggested by Sommerville [128]. Because Sommerville explicitly defines software engineering as computer-based systems engineering, we show his classification in this Section. As we already mentioned above, his classification could also largely be accommodated as a further decomposition of the *reference requirement* type defined by the ANSI/EIA 632 (Figure 3-2). We refer to [128] for some more details on the definition of these terms. However, Sommerville provides only informal definitions for product, organizational, and external requirements and none at all for their subtypes.

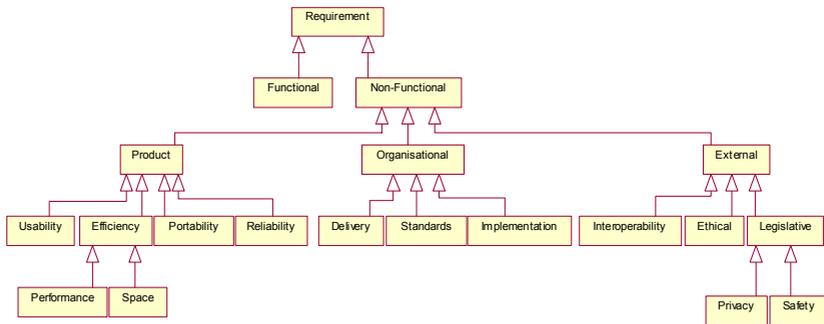


Figure 3-3: Sommerville’s Types of Non-Functional Requirement (Source [128])

3.2 The Software Engineering Viewpoint

In the context of software engineering, properties related to software-intensive systems are usually classified into functional and non-functional ones. As mentioned in the introduction, the notion of non-functional properties has a specific connotation in the software- and partly the systems engineering community. It became to stand for all desired or exhibited properties that exceed a software systems main input/output behavior in its execution environment; i.e. they exceed the system’s main functions. Consequently, non-functional properties cover a rather diverse range of phenomena embracing both the *execution characteristics* in the deployment environment of the software (with high-level properties such as security, performance, etc.) and the *software development economics* (maintainability, etc.). Hence, a common understanding is that the former (execution characteristics) are observable at software system runtime, thus often referred to as “run-time qualities”, while the latter (software development economics) are not. They relate to the economics of building and evolving a software system and its artifacts and are focused on shortening time-to-market and on decreasing development, maintenance, and non-conformance costs. Hence, these properties deal with phenomena that are observable over the product development life cycle or even the life cycle of several products (such as maintainability, reusability, etc.). They are thus also called “development-time-qualities”.

To illustrate this distinction more concretely, we classified some non-functional properties along this thread of thinking in [111] and represented them as depicted in Table 5. The property definitions can be found in Appendix A, Table 25.

Table 5: Non-Functional Properties Observable Over Distinct Life Cycle Phases

Observable over product development life cycle (but not at runtime)		Observable at product runtime	
Main Category	Subcategory	Main Category	Subcategory
Testability		Useability	Accessibility
Portability	Mobility ^{*)} Nomadcity ^{*)}		Administrability Understandability
Integrability	Composeability	Dependability	Availability
	Interoperability		Degradability
Adaptability	Durability		
Openness	Reliability		
Maintainability	Evolvability		Stability
	Extensibility ^{*)}	Survivability	
	Modifiability	Fault tolerance	
	Changability	Security	
	Upgradability	Safety	
Reuseability	Tailorability	Performance	Responsiveness
Deployability	Distributeability		Accuracy
	Configureability		Footprint
	Installability		Schedulability
^{*)} depending on definition, observable at runtime too		Scalability	
		Coherency	
		Timeliness	
		Integrity	

In general, all the properties exceeding the system’s main input/behavior, i.e. its main services for direct users, are inconsistently called one of the following: Ilities [47, 78], non-functional properties (in most of the literature), afunctional qualities [31], extra-functional properties [55, 125], or simply

quality attributes⁴. While some literature and their authors carefully distinguish between these terms or, more frequently, introduced a certain term to stand for some classes of properties only, other sources (and this is the vast amount of recent literature) use them almost synonymously.

In general dictionaries [87] [84] the terms attribute, property, characteristics, or quality are largely treated as synonymous, which is also the case in general philosophy (discussed in Section 4.2.1.1). We posit that in software engineering, however, the distinction between properties and qualities (or quality attributes) is made because of the underlying assumption that the notion of a quality is related to some explicitly or implicitly stated desired. We use bold font to support our claim in the citation of some selected definitions for (software) quality, found in standardization works or in computer science and software engineering specific dictionaries:

- *Quality*: The totality of characteristics of an entity that bear on its ability to satisfy **stated or implied needs**. ISO/IEC 9126-1 [63]
- *Quality*: The totality of features and characteristics of a product or service that bear on its ability to satisfy **stated or implied needs**. Not to be mistaken for "degree of excellence" or "fitness for use" which meet only part of the definition. FOLDOC [56]
- *Quality*: Same as FOLDOC (see above). In addition: "The quality of a system is the evaluation of the **extent to which the system meets the above mentioned features** [functionality, reliability, usability, efficiency, maintainability, portability]." Computer Science Dictionary - Software Engineering Terms [91]
- *Quality Attribute*: "A feature or characteristic that affects an item's quality. In a hierarchy of quality attributes, higher-level attributes may be called quality factors, lower level attributes may be called quality attributes. The features and characteristics of a software component that determine its **ability to satisfy requirements**." Computer Science Dictionary - Software Engineering Terms [91]
- *Software quality*: Software quality is the degree to which software possesses a **desired** combination of attributes. [...] Defining software quality for a system is equivalent to defining a list of software quality attributes **required** for that system. IEEE Std. 1061 [60]

Common to these definitions is the relation of quality to some desires or needs. Further, some means of degree of satisfaction is directly or indirectly expressed. Hence, specifying the desires (in the form of properties) seems to be important in the first place and the explicit use of the term "quality" indicates that some sort of evaluation against the specified desires is to be carried out. As we shall see in Section 4.2.1, this is in contrast to the philosophical definitions of properties or qualities of things, in that they have no conceptual relation to stated needs.

⁴ Note however that the ISO9126 (see Section 3.2.1.1) includes functionality into the term quality attribute.

3.2.1 Quality Models and Requirements Classifications in Software Engineering

In this Section we discuss quality models, and in particular the ISO/IEC 9126-1 quality model, as a means to introduce the terminology and the kinds of properties defined in software engineering.

A quality model is intended as a basis to agree on a definition and on the relationships of the diverse terms used in software engineering in the context of quality of “software”, which also includes perceived properties of a software-intensive system to which software is only contributing in parts. The primary purpose of quality models is to bring structure into quality-related system specifications, in particular requirements specifications, and therefore aid the specification and evaluation process of software products. This again confirms the “satisfaction of needs”-connotation of quality (as discussed in 3.2 above). The general assumption is that specifying quality amounts to specifying a list of required quality attributes; evaluating or making a statement on the quality of a product amounts to listing measured (or predicted) values for quality attributes and relate them to the specifications or some form of valuation scheme.

All known quality models acknowledge the fact that most interesting high-level quality attributes (say maintainability) have no direct single measurement and can be evaluated indirectly only, i.e., through a number of more or less objectively measurable attributes, which in combination have an influence on such high-level quality attributes. To this end, quality models conceptualize the wide range of quality attributes as a hierarchy of determinables and determinates. A determinate attribute (e.g. analysability) is a more specific version of a determinable (e.g. maintainability). The hierarchy of determinables and determinates is relative and generally expected to bottom out in completely specific, absolute determinates that can be measured objectively. Because objective measurement is not always possible, subjective grading and importance weight factors are usually factored into deriving quality statements about determinables.

As the most prominent representative for an agreed upon, explicit quality model we present the ISO/IEC 9126-1 in the next Section.

3.2.1.1 ISO/IEC 9126-1

The ISO/IEC 9126-1:2001 standard [63](called ISO 9126 in the sequel) describes a quality model for software products. Software product is defined rather loosely as “the set of computer programs, procedures, and possibly associated documentation and data” and software as “all or part of [a software product]” ([63], Annex B). Defining an explicit quality model was considered a major improvement over the implicit model that was assumed in the 1991 version of the ISO 9126 standard. The model proposed by the ISO 9126 can be seen as a successor of the frequently cited early models proposed by McCall [80] and Boehm [21].

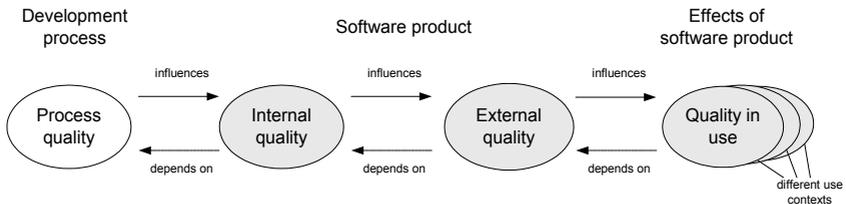


Figure 3-4: Basic Rationale behind the ISO 9126 Quality Model (Source [63])

The underlying assumption or rationale of the model framework is depicted in Figure 3-4. It assumes that the software development process qualities (left ellipse in Figure 3-4) influence the Internal quality of a Software product. The Internal quality has in turn an influence on the External quality of a Software product. The External quality finally influences the quality related to the use of this product (“Quality in use”). As an example for this dependence consider the following: “having source code reviews” (a Software development process quality) influences the source code in that “the number of not initialized variables” (an internal quality attribute of a software product) is minimized, i.e. hopefully reduced to zero. This positively influences the availability, e.g. measured in “mean time between system failures” of the system (an external quality attribute of a software product). Finally, improved availability helps to improve the productivity in the context of the company using the software product, e.g. measured in number of orders processed per month (a Quality in use attribute).

With respect to Figure 3-4, the ISO 9126 quality model defines a consistent terminology and a high-level classification for *Internal quality*, *External quality*, and for *Quality in use*. Figure 3-5 gives an illustration of the abstract terms used in the standard and could be considered its meta-model. Table 6 and Table 7 could therefore be considered instances of this meta-model. Note, ISO 9126 (Part 1) defines a classification hierarchy and a defined set of types of Quality Characteristics and Subcharacteristics, but it does not define the actual set of Quality Attributes and their Metrics.

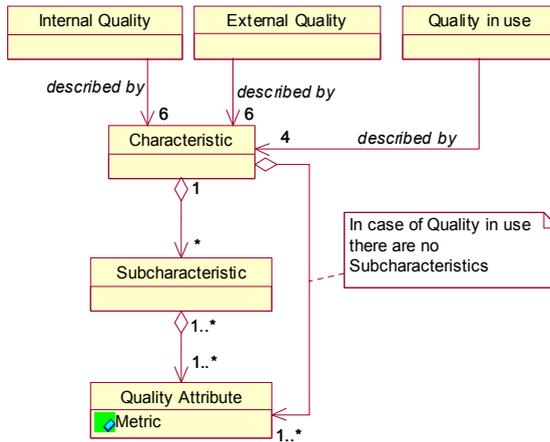


Figure 3-5: General Concepts of the ISO/IEC 9126-1

Quality attributes pertaining to internal and external quality are subdivided into the same six quality Characteristics, each of which has a number of quality Subcharacteristics (see also Table 6⁵). Below the Subcharacteristics we find Quality Attributes. They are the measurable, quantifiable properties of a software product. The latter also includes all its intermediate development artifacts. Quality attributes that refer to the internal quality – internal quality attributes – are typically applied to intermediate deliverables at certain development stages (e.g. attributes of a design specification, source code, etc.). Internal therefore has the connotation of “development *internal* view”. External

⁵ For brevity, we included the definitions of the subcharacteristics of functionality only, just so that the reader can develop an intuition about the level of detail and specificity of the provided definitions. A complete list of definitions can be found in Appendix A Table 25.

quality attributes on the other hand are the measurable properties of a software product that pertain to the behavior of the software product in a defined execution context.

Table 6: ISO/IEC 9126-1 Definition of Characteristics and Subcharacteristics for Internal and External Quality (cf. Figure 3-5)⁶

Characteristic	Subcharacteristic
Functionality - The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions.	Suitability - The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives.
	Accuracy - The capability of the software product to provide the right or agreed results or effects with the needed degree of precision.
	Interoperability - The capability of the software product to interact with one or more specified systems.
	Security - The capability of the software product to protect information and data so that unauthorised persons or systems cannot read or modify them and authorised persons or systems are not denied access to them.
	Functionality compliance - The capability of the software product to adhere to standards, conventions or regulations in laws and similar prescriptions relating to functionality.
Reliability - The capability of the software product to maintain a specified level of performance when used under specified conditions.	Maturity
	Fault tolerance
	Recoverability
	Reliability compliance
Usability - The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions.	Understandability
	Learnability
	Operability
	Attractiveness
	Usability compliance
Efficiency - The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.	Time behaviour
	Resource utilisation
	Efficiency compliance
Maintainability - The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.	Analysability
	Changeability
	Stability
	Testability
	Maintainability compliance
Portability - The capability of the software product to be transferred from one environment to another.	Adaptability
	Installability
	Co-existence
	Replaceability
	Portability compliance

⁶ Further definitions of subcharacteristics are found in Appendix B.

If we look at Table 6 and use the terms of functional vs. non-functional properties, all but suitability, which hints towards the plain functionality provided by the system, would be called non-functional. But note, this definition is relative to what is considered the software product at hand. For instance, a non-functional property at the level of a system might turn into a functional property for a component of that system (discussed in [111] and later in Section 5.2). Also, a specific quality attribute defined for a certain subcharacteristic might only make sense for a certain type of software product.

The quality in use classification only knows one level of characteristics with four defined types (see Table 7). Quality in use refers to the extent to which a software product meets the needs of specified users in a specified context of use and as such shall be related to user goals and tasks. It is the end-user's perception of a system's quality with respect to the task the user has to fulfill.

Table 7: ISO/IEC 9126-1 Quality In Use Characteristics
(cf. Figure 3-5)

Effectiveness - <i>The capability of the software product to enable users to achieve specified goals with accuracy and completeness in a specified context of use.</i>
Productivity - <i>The capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use.</i>
Safety - <i>The capability of the software product to achieve acceptable levels of risk of harm to people, business, software, property or the environment in a specified context of use.</i>
Satisfaction - <i>The capability of the software product to satisfy users in a specified context of use.</i>

In general, characteristics or subcharacteristics have quality attributes associated with them (refer again to Figure 3-5). A quality attribute is a measurable physical or abstract property of an entity. A *metric* defines the measurement method and the measurement scale. The measurement is the actual act of using the metric and assigning a value to an attribute. The standard's Part 1 does not define which attributes and metrics a certain (sub-)characteristic must have. Substantial research work and its results are available, which aimed at defining such quality attributes or metrics. A good overview is found in [45]. For the purpose of providing standardized metrics, Part 1 will be enhanced with Parts 2, 3, and 4. These parts are intended to standardize a basic set of metrics for external, internal, and quality-in-use characteristics, respectively. They were not available during the period of our research.

During the finalization of this thesis report, however, first drafts (not yet standards) of Parts 2 and 3 were released [64, 65]. Part 4 is still not available.

3.2.1.2 IEEE Std. 830

In order to also present a standardized classification of *software* requirement types, similar to what we did for *systems* requirements in Section 3.1 above, we briefly present the classification of the IEEE standard "Recommended Practice for Software Requirements Specifications" [59]. In Figure 3-6 we again add the, partly shortened, definitions as found in the standard into UML notes. While it is not as detailed as for instance the ISO 9126, this classification covers in principle all the categories of the ISO9126. In its breadth or diversity of properties it even goes somewhat beyond the ISO 9126. The IEEE 830 categories *Functions*, *Performance Requirements*, and *Software System Attributes* can

relatively easily be mapped to counterparts in the ISO 9126. The *External Interfaces*, *Design constraints* and *Logical Database Requirements* are more specific to the IEEE 830. They have a less direct mapping to the ISO9126. What is described in the IEEE 830 standard document under *Design Constraints*, however, matches to the ISO 9126 “standards compliance” subcharacteristic, which is present in every characteristic (recall Table 6). Requiring specific *External Interfaces* and their properties can be considered an additional solution constraint on realizing the required functionality (or Functions in IEEE 830). In the ISO 9126 this is not present because it is assumed as a precondition for, or subsumed under, suitability. Finally, the *Logical Database Requirements* are very specific to the IEEE 830 in that they shall specify particular requirements on the data that is to be placed into a database (should the system to be developed contain one). The requirements would essentially reflect a database schema or data model, integrity constraints, and accessing capabilities. Interestingly, the IEEE 830 is not further elaborating on this category at all. This category does also not appear in their templates for requirements specifications.

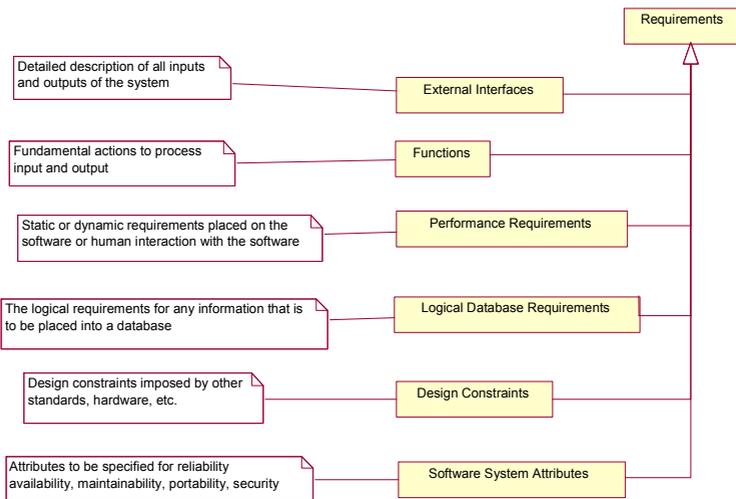


Figure 3-6: Requirement Types as Defined by the IEEE Std. 830

Summary of Part I

In Part I we provided the context of our work. We presented the disciplines of software- and systems engineering and their current notions of systems and properties, as well as some prominent representatives for development methods in both areas. While most of the methods are mature they fall short in one or several of the following abilities: support for more than the design of functional properties, model-based traceability of property realizations, modeling of arbitrary hierarchies of concrete and conceptual systems, and uniformity to seamlessly integrate systems- and software systems modeling.

In Part I, we also presented the current understanding and breadth of properties as they are found in systems- and software engineering. We conclude that there is a lack of a fundamental, general account of non-functional properties, in fact of all properties. This is evidenced by the diversity of definitions, by the diversity of classifications, and by the fragmented approaches to deal with properties in development methods for systems. Such an account should help to identify what properties are relevant given a certain system, how to conceptualize and represent them, and how to trace the design for them in the progression of development steps. All this requires a basic model of systems and properties. This is the subject of and proposed in Part II.

Part II

Methodological Building Blocks In Dealing with Hierarchical Systems and their Properties

Part II elaborates on the foundations of properties and systems from the perspectives of basic philosophy, cognitive science, and systems science. It defines the fundamentally different types of properties and of systems and thereby explains the confusing term of a non-functional property. Part II shows how the gained insights and our interpretations are used in a conclusion-oriented mode to formulate basic tenets, i.e. rules which we deem important for modeling and understanding properties and systems. Further, it presents the conclusions that lead to our proposed conceptual models for *properties* of components and for concrete *systems*, respectively. The combination of the two yields our holistic model for properties of concrete systems. The tenets together with the holistic model of systems and properties represents our philosophical underpinning for all the methodological building blocks and applications discussed thereafter. This includes (a) the 2-2-2 model as a simple aid to discover modeling-relevant perspectives on a system for the purpose of identifying different classes of stakeholders and thus different classes of properties, (b) basic patterns of model-based property traceability through hierarchies of systems, and (c) UML-based representations for systems and properties, and for property traces.

4 Philosophy of Properties and Systems

Objectives: This chapter introduces the fundamental nature of properties and of systems and presents our interpretations and conclusions. Our conclusions are formulated as individual tenets in support of conceptualizing properties and systems. They are also formulated in what constitutes a model ontology, namely a suggested general model of properties and systems. In order to be able to express our conclusions in the form of models, it is a further objective of this chapter to clearly define the notion of modeling and its related concepts.

After having read this chapter, the reader will:

- know our distinct terminology for the process of modeling and for the concepts to describe this process and its results;
- understand the basic distinction between functional and other properties, which is derived from the philosophical notions of primary and secondary properties, and the cognitive distinction between essential and non-essential properties;
- understand the difference between concrete and conceptual systems and the value of this distinction in systems- and software engineering;
- know our general, conceptual models of both properties and systems;
- know our general, conceptual model of general systems *and* their properties, which results from the combination of the general model of properties and the general model of systems.

Before we proceed to our philosophy of properties and of systems, we elaborate on our understanding and our terminology related to modeling. In particular, we explain our use of the terms reality, perceived reality, conceptualization, and representation. They are fundamental not only to defining our notions of viewpoint, agreed conceptualization and agreed representation but also to positioning important forthcoming Chapters and Sections of this report. In particular, the role of the knowledge generated in the subsequent Sections on Properties (4.2), on Systems (4.3), and on the combinations thereof (4.4) can be explained with respect to our overall goal of understanding the different natures of properties of systems with the objective to model them.

While we have followed the sequence systems and then properties in the previous Chapters, we purposefully reverse the order of discussion in this Chapter for an important logical reasons: First, we want to elaborate on properties of *things*, where thing is the mundane and most general term for anything that is perceived through our cognitive abilities. Second, we want to constrain things to systems and as such constrain the universe of properties to those that are relevant for systems.

4.1 Modeling

Philosophically speaking, thinking is modeling [134] and thus, agreed upon or compatible models are key to “inter-think”, i.e. they are key to facilitate communication among humans, among computers, and among humans and computers. As evidenced by the model-driven development methods (recall Section 2.4), model representations are central in any team-based effort that aims to design anything but trivial artifacts.

Modeling is fundamentally a cognitive process in which knowledge is structured through our capability of forming abstractions over some real, hypothetical, or imaginary phenomenon. As an example, Figure 4-1 shows an observer who is attending to a real phenomenon (her *perceived reality*) as part of some “reality” or universe of discourse (Figure 4-1 (a)).

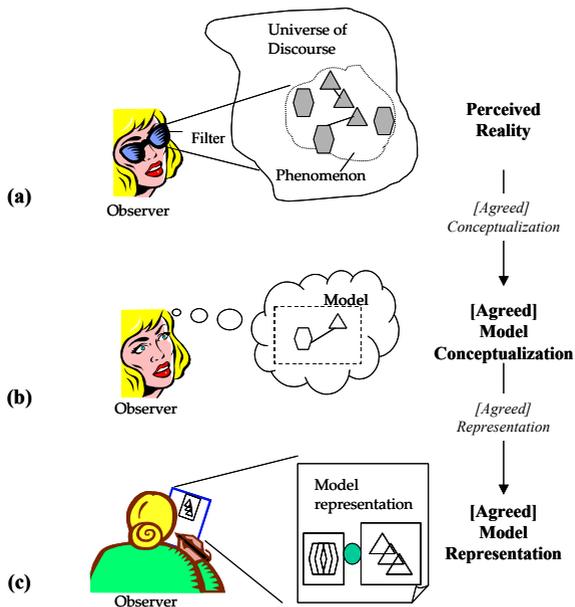


Figure 4-1: From Perceived Reality to Model Representation

Her mental model is the observer’s or modeler’s conceptualized understanding, i.e. an understanding organized around concepts and concept relationships. The cognitive process of *Conceptualization* leads to the end result: the conceptualization. We call the latter *Model Conceptualization* only in Figure 4-1 (b)⁷ and whenever the context of discussion leaves room for ambiguity between whether the process or the result is meant. Otherwise the use of the term conceptualization or even model is good enough. The model conceptualization is personalized because

⁷ Note, we could have used *model* instead of conceptualization. But because of a (software) engineer’s association of the term model with a (graphical) model representation, we consciously use conceptualization instead of model, when we want to emphasize our understanding or the meaning of a phenomenon (or a theory) rather than its representation.

it results from applying an observer specific, conscious or unconscious, filter to the formation of knowledge about the phenomenon. The conceptualization and therefore the filter are determined by the goal of the modeler. We subsume the goal and thus the filter under the notion of a viewpoint, which stands for “the mental position from which things are viewed”[87]. This definition of viewpoint is perfectly to the point, because goal as well as formation of knowledge refers to mental issues. Each observer may have one or more conceptualizations of the phenomenon. The multiplicity largely depends on whether we accept the idea, which we do, that a single observer has the cognitive abilities to mentally partition the knowledge about phenomenon.

Finally, if the observer/modeler wants to share her model conceptualization or record it for later use, she needs to create a *model representation* (Figure 4-1 (c)) through the process or the act of *representation*.

Now, if we want to share our personalized conceptualizations with someone else (for example to perform some collaborative task) we need a shared understanding of the purpose of these models, an *agreed upon* conceptualization, and an *agreed upon* representation. *Agreed* conceptualizations lead to *agreed* model conceptualizations (Figure 4-1 (a) to (b) with its emphasis on [Agreed]). *Agreed* representations lead to an *agreed* model representation (Figure 4-1 (b) to (c) with its emphasis on [Agreed]).

We need agreed conceptualizations because the concepts used in the model conceptualizations (concept types, relationships to perceived entities, etc.) are still modeler-specific, but we need to agree on them among a group of modelers. In other words, we need to synchronize our thinking and understanding of phenomena. When a modeler represents her reality, she is using her knowledge and her perception to decide what to model. In other words, what a model represents is tightly linked to the discipline and the experience of the modeler [28]. Modeling in system engineering typically involves multiple disciplines. To be able to relate the models made by the specialists of the different disciplines, it is essential that they agree on what entities they consider as relevant in the reality and how these entities are modeled in the possibly different models. In other words, they need to *agree* on the “conceptualization” of their reality. The term of “conceptualization” comes from Tarski’s theory of truth [132]. According to Tarski, a fact is true for a group of people if there is an agreed conceptualization between all people that the fact is actually true. In other words, truth does not exist in the vacuum and can be considered (at least in the context of systems modeling) as a social agreement between all people involved in the same undertaking. As a consequence we propose to name the relationship between the reality and the model “conceptualization”. Others, e.g. OMG’s MDA, use the term “abstraction”, which we recommended to change in MDA in order to reduce ambiguity [141].

We need agreed model representations to transfer our understanding. As observed by Peirce [53], any representation has four essential aspects: (1) it is realized by a representation bearer; (2) it has content or represents one or more objects; (3) its representation relations are somehow “grounded”; and (4) it can be interpreted by (will function as a representation for) some interpreter. As an example: A UML class diagram might be drawn on paper (bearer), its content might depict a type hierarchy of animals, its representation relations are based on graphs with defined box and edge types: box stands for animal type, edge stands for “is-a” relationships, and it can be interpreted by another human (given an agreed upon conceptualization and representation).

Only when we have both agreed conceptualizations and agreed representations can we turn the model from a personalized mental description into an impersonalized organized description of knowledge about a phenomenon ready for communication.

For the conceptualization and representation relations to be agreeable, an explicit *modeling ontology* and *model notation* is needed, respectively (see Figure 4-2). A modeling ontology is a representational vocabulary for a shared domain of discourse. This definition of modeling ontology therefore hints at its destined use, namely to define what exists or will exist (“ontology”) in models (i.e. in conceptualizations and their representations). In our case, such a modeling ontology uses the

result of our philosophy of systems and properties. All concepts used in the conceptualization of a system and its properties are therefore instances of our general concepts. The notation for our conceptualizations depends on our extension of the UML. The references to the respective Sections in this thesis report are given in the UML notes of Figure 4-2. In order for the modeler to create representations that are interpretable by others, they (including the modeler) should know the agreed conceptualization and agreed representation. These relations (*create* representation, *know* modeling ontology and *know* notation) are not shown in Figure 4-2 to not overload the figure.

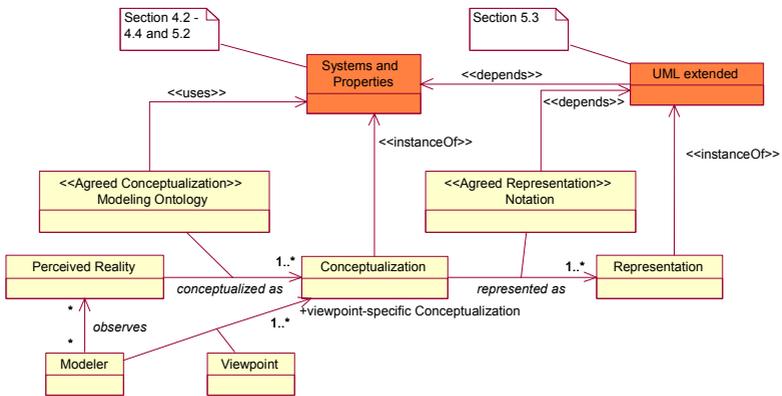


Figure 4-2: Domain and Modeling Ontology and Their Dependencies on Our Work

A visually more intuitive picture of the relationships between reality, its models, and the use of general (or meta) concepts is shown in Figure 4-3. To explain the figure, consider the following example: A modeler observes a few woody perennial plants in her perceived reality. These plants are instances of the general ontological concept of a *stable entity*. In her conceptualization, she conceptualizes these entities as *trees* because of her associative memory and the types of properties she ascribes to trees. Based on a further criterion - the physical proximity of these trees - she conceptualizes the sum of the trees as a *forest*. On the general- or meta level, her conceptualization is that of a *system* (the forest) as being a set of interrelated *components* (the trees). Finally, for the purpose of representation, she decides to draw a diagram that shows the concept of a forest as being a *package* consisting of trees represented as *classes*. Hence, she followed an agreed notation.

The question might arise as to why these many indirections are needed (i.e. why are the concepts in reality different to the concepts in the conceptualization and again to the representation). The answer is that we need this distinction to be able to express relationships between them. For instance, we want to express that a component (in the model) has an abstraction relationship to a stable entity (in reality); or, that a particular individual stable entity (say a human) is found twice in the model, once as a component and once as a system.

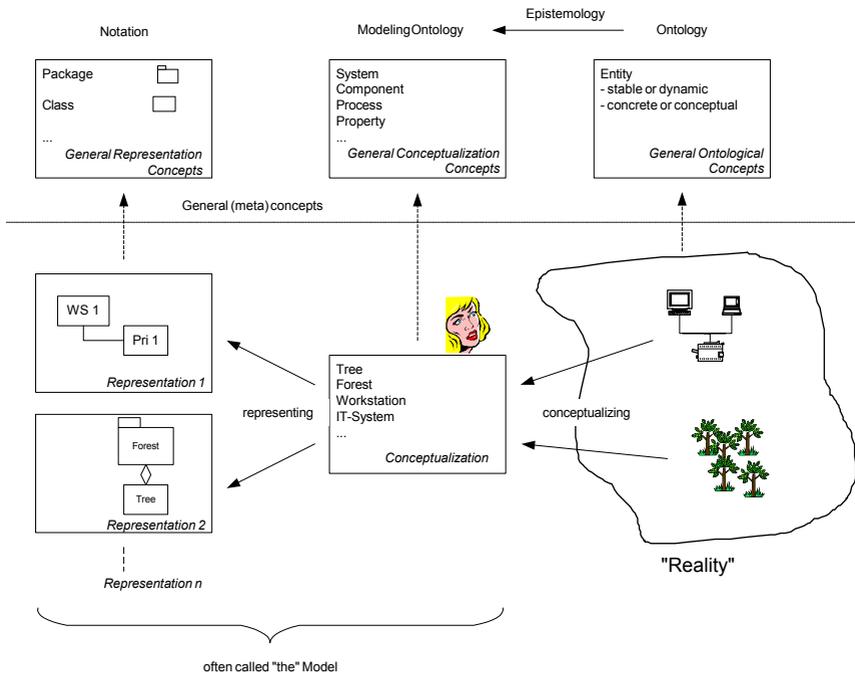


Figure 4-3: Reality, Model, and General (or meta) Concepts

The previously stated modeling fundamentals were discussed in parts in similar ways by other researchers in various branches of computer science [61] [66] [27] as well as in other scientific disciplines such as cognitive science [73] or systems science [10]. In several works (among others in OMG’s Model-Driven Architecture, MDA [95]), however, the model representation is considered “the model”. This presents a short cut from phenomenon to model representation. It is acceptable, given there is an implicit agreed conceptualization with the representation, but it has the risk of underestimating or overlooking the importance of an agreed model conceptualization [141], and thus the potential for watering conceptualization and representation issues, or even of building precise representations on imprecise agreements of what to model.

We should also note that in software architecture the notions of viewpoint and view (sometimes not distinguished as in the MDA, sometimes distinguished as in IEEE Std. 1471-2000 [61]) are a vital concept to structure model representations of systems. View is defined as “a representation of a whole system from the perspective of a related set of concerns” [61] and viewpoint as the techniques and languages to construct and use views. This definition of viewpoint, which we do not use, is specific to software architecture and different from our general definition given above. However, since concerns are related to stakeholders, the use of views is related to representations that address concerns specific to stakeholder-types. This is in line with our, cognitive science driven, definition of a viewpoint-specific model. We show the implications of a viewpoint when we introduce view as a representation concept in Section 5.3.1.

Let us finally remark that assessing the quality of a model representation requires knowledge about the intended audience and the purpose of the model. In general however, a good model and its associated representation is:

- (a) one which is based on agreed upon model concepts *and* representation concepts because, if not more, it can be understood by the agreeing parties,
- (b) one which can be efficiently processed by humans, i.e., it must be compatible with our cognitive capabilities to perceive information, process it, and form knowledge,
- (c) one which, if so needed, can be efficiently processed by computers, i.e., it must be compatible with the computer's capabilities to "perceive" information, process it, and deduce "knowledge", and
- (d) one that serves the purpose of the model in the first place, which usually means that we want to be able to reason about an existing or to-be-built phenomenon.

4.2 Properties

The challenge in our work is not concerned with the theories behind individual properties (such as what is *green*, or what is *having a latency of*, or *what is security*). It is rather concerned with how we can generalize our understanding of the notion of property or its synonyms to a level where we can suggest a principled manner to model them in the context of systems and where we can suggest a principled manner to identify the relevant properties in a given context. Our basic research on properties is summarized in this Section as well as our hypotheses for such a general model of properties.

Note, the terms thing, object, and entity are frequently used interchangeably in other literature, mainly because of linguistic reasons and its reflective nature in defining terms. We use the term *thing* as a linguistic feature in that it stands for the most general placeholder for anything not precisely designated yet. It can only be understood in the context of discussion. We use entity, however, in its specific etymological sense, which includes the notion of existence, i.e. an entity is something that has separate and distinct existence and objective or conceptual reality [84]. While entity may refer to dynamic and static things, we use object to denote static things only⁸. Hence, entity is a generalization of process and object.

4.2.1 The Philosophical View On Properties

We looked into philosophy to find very basic, universal or foundational, definitions for qualities (or properties to use the more neutral term) of any kind. Hence, the basic question to be answered was: *What are qualities/properties of things?*

4.2.1.1 What are Properties Good For?

Coming to grips with properties is pervasive in philosophy. Plato's theory of Forms ([120] p.93) (where Form is said to be Plato's term for property,) seems to be one of the earliest accounts on what is today called properties. The term property includes attributes, qualities, features, or characteristics of things. It even encompasses relations such as *being faster than*.

The need for properties is motivated by their explanatory roles they have to fill. They came into being to describe phenomena of *interest*, especially in those cases where the name of a phenomenon did not or does not yet exist or where it is all too general so as to allow all kinds of ambiguities. Properties in general shall help explain such notions as recurrence and resemblance of groupings of

⁸ Static is of course relative. In phenomenal perception, an object is perceived as being static against its background. This is further discussed in Section 4.2.2.1.

things (including comparisons), or recognition and classification of (new) things. A stakeholder is a role that represents groups of people who have similar interests in the same phenomenon. These people are said to have stakes on the phenomenon. Thus, the choice of properties and their importance is clearly related to certain stakeholders or stakeholder classes.

From an ontological viewpoint, the existence of properties is determined empirically. That is, properties and their definitions are invented by humans and *there is no a priori, logical or conceptual method to determine which properties exist* [130]!

Tenet I. Any property is an invention by humans. There is no a priori method or logic to determine which properties exist. Hence, the set of properties must be defined and agreed upon in any universe of discourse that is shared by more than one observer.

From a natural language viewpoint, there is no single idiom to talk about and use properties. In other words, properties are distinct from their representations and *the same property may have different representations*. In the English language for example, properties can appear as single terms with any of the many suffixes such as ‘-ity’, ‘-ness’, ‘-hood’, ‘-kind’, ‘-ship’, etc. (e.g. as in ‘safety’), or as predicative expressions in multiple ways (e.g., ‘executes safely’, ‘is safe’). Hence, since properties are inventions by humans not only their meaning need to be defined but also their possibly numerous, not only natural language-based, representations.

Tenet II. Any property can have a number of representations. The possible representations for every defined property need to be agreed upon.

As with any knowledge-related thing, humans tend to categorize also properties, i.e. we describe properties by means of “meta-properties” or inherent characteristics of certain categories. We define two such inherent characteristics that are important for us: *complexity*, and *specificity*.

- **Complexity:** Properties can be *simple* or *compound/complex*. A complex property is always conceptual and can only be understood if the theory behind forming the complex property is defined. As an example for a complex property we can think of any property that is the logical structure of properties. E.g. ‘being my grandfather’ implies that the person this property is ascribed to is ‘male *and* older than I am’. Or ‘being CMM Level 3 certified’ implies that the software development unit this property is ascribed to ‘has a software process that is documented and standardized, and that all software projects use an approved, tailored version of this standard software process for developing and maintaining software’[123].
- **Specificity:** A property can be a *determinable* or a *determinate*. The distinction, however, is relative. In essence, a determinate property is a more specific version of a determinable. For example, “up-time” is a determinate property (i.e. a more specific one) of the determinable “availability”. The measure “length of time between failures” is in turn one possible determinate of “up-time”. The hierarchy of determinables and determinates is generally expected to bottom out in completely specific, absolute determinates. In software engineering, such leave determinates would be called quality-carrying properties, or direct properties, or tangible/measurable properties, to name a few.

In software engineering, the combination of these two characteristics is the basis for the *decompositional* approaches of quality models (Section 3.2.1 or [45] p.338). They use a hierarchy of determinables/determinates, where each determinable is usually a complex, i.e. a compound property. Simply said, the representation of such a quality model results in a tree with spanning level greater than one for every but the leave nodes.

4.2.1.2 Primary and Secondary Qualities

A very important philosophical distinction of properties is the one that tries to separate fundamental properties of concrete objects - primary properties - from others, with the idea that we

can explain why things have these other properties - secondary properties - by consideration of their primary ones. Because of Locke's impact on philosophy and his using of the term *qualities*, primary and secondary properties are often discussed under the terms primary and secondary qualities⁹.

Qualities or properties (to use the non-Locke biased term) of concrete objects belong to one of the following categories:

1. **Primary properties**, which are *physical properties* or logical constructions of physical properties.
2. **Secondary properties**, which are *dispositions*.

Physical properties of concrete objects would typically correspond to the quantities in physics such as length, mass, etc. with their respective units such as meter, kilogram, etc. Thus, they essentially refer to what in physics is found under the International System of Units (SI) [93].

Logical constructions of physical properties are conjunctions and disjunctions of physical properties. For example, weighting 25kg +/- 0.5kg is a disjunction. The term *primary quality* subsumes both physical properties and logical constructions thereof. The philosopher's intuitive idea was that primary properties/qualities are *objective* features of objects in the world. For completeness, let us mention that some philosophers restrict the primary qualities to only those physical properties and logical constructions thereof that are fundamental and inseparable from all matter (point in time and space, mass, charge, and spin). For us, the less restricted definition is good enough.

Tenet III. In order to be able to reason about values of primary properties we need to adopt an instance level modeling approach. Property roll-up (determining the value of a primary property of a system given its constituent components) and budget-down (specifying values for primary properties of constituent components of a system) requires aggregate instances, i.e. complete aggregates.

A *disposition* is the potential or tendency of an object to act or react in a specific way under certain conditions. For example: ice does melt if the temperature of its environment is above 0 degrees Celsius; a pullover is red if seen under special light conditions; the monitor will resume working from its energy saving mode if any key is pressed on the keyboard. We therefore define a disposition as a behavioral property of an object, where behavior is defined as the response to stimuli. At a closer look, dispositions of concrete things are powers of objects to produce sensory experiences in humans, i.e., *a property of the nature of a disposition is relative to the observer and the condition during which it is observed*. A disposition to produce sensory experiences is called *secondary property/quality*. Russel [120] coined the term "sense-data" to more explicitly refer to the sensory experience properties. The important reason why we must identify an object for the sense-data is our wish to agree on the same object by different people.

A very interesting philosophical concept, as part of the explanation for what is a disposition, is the *base* of a disposition. In essence, it says that the base of a disposition is responsible for the manifestation of the disposition. A base is the "underlying" structure of an object that causes the object to manifest a certain property the way it does it. For example, the base for the fragility of a glass is its molecular structure. "Underlying structures" implies a component/composite construct and suggests that we conceptualize an object as a system. This implies a number of levels (at least two). The number of levels for a given system depends on whether an abstraction (of a property) at a certain level is good enough (and agreed upon by as many people as possible) for reasoning at higher levels.

⁹ The distinction of primary and secondary properties are credited to philosophers such as Locke, Holbach, or Hume, but essentially derive from the ideas of Robert Boyle and even date back to works from Descartes. Good references are [6], where Averill reports on "qualities" and [145], where LeBuffe reports on Holbach's theory of matter and Uzgalis on John Locke's work.

Tenet IV. Secondary properties refer to behavior of an object, i.e. they are the response to a stimulus. Behavior is realized by the underlying structure of parts. Declaratively specifying a secondary property amounts to describing the response to a stimulus considering the context and the observer. Realizing a secondary property amounts to designing an underlying structure of interacting components.

4.2.2 The Cognitive View On Properties

Cognitive science is an interdisciplinary science that combines computer science with linguistics, psychology, anthropology, and philosophy. In general, it tries to understand what a conceptual system is and how a human’s cognition system works. Our reason for looking into cognitive science was to understand how humans perceive concrete and conceptual things, how they ascribe properties to them, and thus how they organize knowledge about them. This understanding shall in turn be used to better comprehend what are good ways for learning, what are good ways to represent knowledge, and consequently, what are good and practical models. Furthermore, we were also interested in finding explanations for the fact that software engineers tend to treat some properties as more important than others, namely functional properties over non-functional ones.

4.2.2.1 Perceiving the World Around Us

Humans (represented by an *Observer* in Figure 4-4 below) perceive phenomena in the world around them as entities that can be distinguished from their environment or background. Hence, environment and background are possible related contexts in Figure 4-4. A context stands for both the discourse that surrounds a language unit and helps to determine its interpretation [87], and - used in a circumstance, physical environment sense - the interrelated conditions in which something exists or occurs [84].

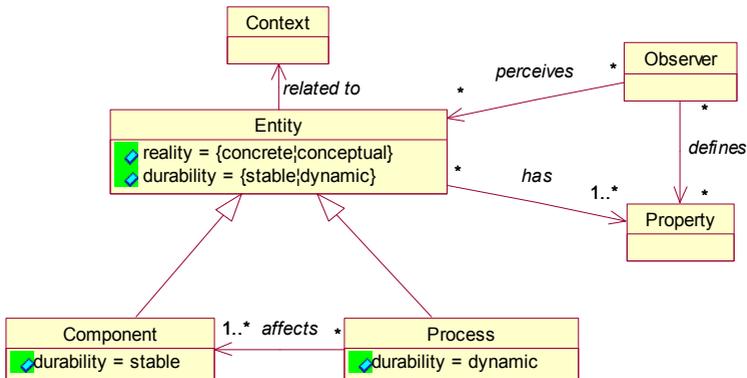


Figure 4-4: Component and Process as Fundamental Entities in the Perception of Reality

Entities are of two fundamental kinds: components or processes. Entities, which we perceive as being relatively stable in contrast to the perceived environment, we call *components*. We could have used the term object instead, but we prefer component to emphasize that such an entity is always related to something (and be it the environment which we deem unimportant at times). Besides components, we perceive occurrences of something happening. Such happenings we call *process*. A process always requires components to be “tangible”, i.e. we cannot talk meaningfully about a process

without mentioning the effect the process has on components. The effect amounts to changing components in some way. This includes the creation of new and the destruction of existing components. In general, the effect is expressed in terms of some properties of the components. These properties are called state-representing properties because their values represent the state of a component at some particular point in time.

Both components and processes can be *concrete* or *conceptual*. We use conceptual, rather than abstract, to emphasize its connotation of being related to concepts, i.e. to ideas, knowledge, or information.

Concrete components (such as a tree, a car, a person, a computer) are physical, i.e. they have spatiotemporal properties and causal powers. They are transformed by concrete processes. For instance, the concrete process of *sawing* refers to the process where a tree is cut into pieces by woodcutters. Or, in the context of an ink jet printer, the concrete process of *printing* refers to the process of “reproduction by applying ink to a paper as for publication...”[87].

Pure conceptual components have no spatiotemporal properties. They exist in mind only. To make them tangible, conceptual components become *representations*. They are representations of knowledge (or information to use a more familiar term). Representations are always the combination of the knowledge bearer, a concrete component, and the intangible meaning of the representation. A more detailed discussion follows in Section 5.1. However, what is of interest of a conceptual component is its meaning. Conceptual components are transformed by means of conceptual processes. Pure conceptual processes in everyday life are cognitive processes such as modeling, categorization, etc. For example, transforming a mental model of a tree (the conceptual component) by means of refinement (the conceptual process) might yield the same component having different or more properties (e.g. a tree with green leaves). Execution of software, when focused on the information processing only, is also a conceptual process transforming conceptual components. For example, in an information system the information component “currentBalance” of a bank account, perhaps represented by a data structure, is changed into a new state through a conceptual process “setBalance”, which might modify a field in the data structure. However, computer-based information processing is never a pure conceptual process because of the supervening transformation of some representation bearer.

It is interesting to mention already here (i.e. before the discussion of systems in Section 4.3), that if an information process is characterized not only from the conceptual- but also from the bearer, i.e. the concrete processing point of view, we move into the realm of some important category of non-functional properties: namely execution-related performance. Hence, **what is considered functional in software engineering normally relates to conceptual processing of conceptual components**, i.e. it corresponds to a pure information processing viewpoint. If what matters is not only the information process but also how this conceptual process is carried out as a physical process, we need to investigate properties of the concrete process too. For example, if a spell checker verifies a document for correct spelling (i.e. it processes a conceptual component), we might be interested how long it takes. This duration is dependent on the concrete process that “carries” or supervenes the conceptual one. Consequently, the duration depends on the media on which the document is carried (memory, hard disk), the length of the document (number of symbols), the processor speed, etc.

4.2.2.2 Cognitive categories

The concept of a *cognitive category* is central to knowledge and human reasoning and therefore also to modeling. In the most general definition, a cognitive category refers to groupings of concrete components as perceived by humans. Humans understand the world not only in individual things but also in categories of things. This goes as far as that we seem to attribute real existence to those categories. Cognitive categories determine which properties we ascribe to an individual member of that category. While cognitive science agrees on the fact that humans organize knowledge by cognitive categorization, there are two schools of thought that attempt to explain this phenomena:

objectivism and *experientialism*. The latter, a relatively young stream, showed that the objectivism paradigm is not enough to understand human categorization. This hypothesis and much of what we say below is derived from our interpretation of Lakoff's excellent treatment of the subject matter [73].

Objectivism defines a cognitive category strictly based on shared properties by *all* its members. It is a set theoretical approach and closely related to the notion of a class or type as used in programming languages. Experientialism says that *not all* members of a category must have attributes in common with all other members and also, that categories and thus reasoning has to do with the characteristics of the organism doing the reasoning and with its *context*. Hence, a category is not something that is objectively in the world; it is *not* disembodied. For a human, learning to categorize is different from learning to use the logic of classes. Object-orientation (OO), as introduced for programming computers, was strictly based on classes as a concept for abstract data types and it had this intuitive "right" feeling also for object-oriented modeling of the world, mostly because classes have a strong correspondence to categories in simple applications. Modeling the world, however, has much to do with cognitive categories and is therefore not as disembodied as classes tend to pretend. The fact that *classes and categories are not the same* is a major reason for OO's insufficiency for more complex projects (examples are found in [50]). If we restrict ourselves to model in class hierarchies we miss important groupings of members that are much more intuitive. Hence, the important message here is that *a category and its highlighted attributes are heavily dependent on the person and its context. There is no objective right set of attributes. Any modeling approach must take this into account explicitly.*

A second difference between objectivism and experientialism is their theory on how categories are formed. Experientialism holds that the natural way of categorization is *not* by forming simple taxonomic hierarchies beginning at the top or at the bottom, as we tend to do in natural sciences and engineering, but rather by constructing hierarchies by starting somewhere in the middle - the *basic level*. Knowledge is mainly organized at basic levels. Objects at basic levels are determined by the overall part-whole perception, i.e., *gestalt*¹⁰ perception. Good examples for this hypothesis are given in [73] (see pp. 47 ff). The overall thesis is that humans list far more attributes, i.e. most of what they know, for basic level categories than they do for super- or subordinate levels. Take as an example "car", which is a basic level category. Compare what you know about it (by listing properties of car) with what you know about its category at the superordinate level (vehicle) or subordinate level (sports car).

4.2.2.3 Basic level categories, and basic level properties

As indicated by objectivism, human categorization is influenced by both *experience* (perception, motor interaction, and culture) and *imagination* (metaphor, metonymy, and mental imaginary). In this context, metaphor refers to a figurative likeness or analogy between members, metonymy refers to reference-point thinking, i.e. a member of a category can stand for the whole category for reasoning purpose. As a consequence of human categorization, reasoning depends on the same two aspects. While experience determines basic level structures, imagination is mainly responsible for super- and subordinate levels. Thus, basic level categories are determined by a correlation of the overall perceived part-whole structure of an object (the *gestalt*) with our motor interaction and with the functions of the parts. The notion of a property is therefore the result of our interactions given our bodies and cognitive apparatus; they are called *interactional properties*. Basic level properties are not inherent to the objects but a result of the way people interact with objects. Since their interaction is through the parts or units of an object they perceive the object at the same time as a whole and as a collection of parts – thus, the mentioned *gestalt* perception. Super-ordinate categories have different types of properties because they do not seem to be characterized by defined images or motor actions. But they do have other human-related properties, like purposes and functions. As an example, consider

¹⁰ Gestalt: a structure, configuration, or pattern of physical, biological, or psychological phenomena so integrated as to constitute a functional unit with properties not derivable by summation of its parts [84].

vehicle as the super-ordinate category of car. We do not have a clear mental image of a vehicle; neither do we have bodily interactions with it. The properties of a vehicle could be related to its purposes (e.g., “easy carrying of goods from A to B”).

Cognitive categorization, and in particular the notion of a basic-level category, plays a central role in modeling complex systems. Why this? Complex systems are essentially hierarchies of systems [126], each level dealing with special kinds of systems, often relating to a special engineering discipline. We find basic-level objects on every hierarchical level. Basic-level objects are mainly the result of experience, as we said above. We associate experience with experts. Hence, modeling of system hierarchies requires explication of basic-level objects and their properties per level as derived from the experts, and then an agreement on how these basic level objects relate across levels.

4.2.2.4 Essentialism and Essential Properties

Essentialists provide, as part of the theory of essentialism, an answer to our question as to why we treat some properties as being more central than others. Essentialism holds that among the properties that things have, some are essential, i.e. they make the thing what it is and without which it would not be that kind of a thing [73]. A property is therefore *essential*, if a thing necessarily cannot exist without being an instance of that property. A property is accidental if a thing can exist without being an instance of that property. For example, an essential property of a printer is its ability or purpose to print text. An accidental or *non-essential* property of a printer is its grey color (it might have been white instead), or that it is wider than tall, etc. We therefore call these two types of properties *defining* vs. *distinguishing properties* to hint towards the fact that essential properties define the type of a thing and non-essential properties distinguish members of that type. A thing can be natural or artificial as well as concrete or conceptual. In Aristotle’s early account on this subject matter, essential properties are simply relating to *what* an object is versus *how* an object is. Interestingly, this very informal definition is still used today to distinguish for instance a system service from a system’s quality of service [1]. Further, several cognitive scientists, who observed that some members are more central to a category and thus their properties somehow more essential than others, confirm this idea of *essential properties*. For basic level categories essential properties would correspond to interactional properties.

In general, the notion of essential property explains why non-functional properties are second-class citizens in software engineering. They do not determine the type of an *information* processing object. Non-functional-properties rather express differences between objects that conform to that type. This can be a difference that relates to how the information is processed (conceptual process) and/or how the bearer of the information is processed (concrete process). For example, the programmatically relevant type of a software component is determined by its syntactic interface definition, where basically a number of operation signatures are defined. While two realizations of that type may conform to this very same type definition, some of the characteristics may differ. For instance, the execution time of certain operations may be different, or it may refer to the source code that may look entirely different, which affects the component’s maintainability. Because our current modeling habits basically live in the world of abstractions¹¹, we abstract away, or omit details of, the differences of objects that are represented by the same abstraction (i.e. the same type). And these differences are largely found in the differing non-functional properties of those objects.

The fact that the essential properties are important properties and often thought to be functional properties is a thesis also supported by scientific realists, a special philosophical strand concerned with common sense scientific research. For example, *being a computer*, *being a power transformer*, *being a database management system* relate to functional properties because it entails to play a certain causal role. For instance, to be a power transformer means to play a causal role in the transmission of electric

¹¹ Most modeling activities show prototypical phenomena or constellations based on a type (or class) concept and type (or class) level relationship. This does not surprise, however, because it is the most natural way of human reasoning. Very few modeling activities stay on the level of instances.

power. It is in principle possible for quite disparate physical mechanisms to play this role. “To say that something exemplifies a functional property is, roughly, to say that *there are* certain properties that it exemplifies and that together they allow it to play a certain causal role” [130]. It is these properties that we call functional and from which the object receives its role name(s).

4.2.2.5 Conclusions of the Cognitive View of Properties

From this discourse into cognitive science we conclude that the most important concepts for knowledge organization are at basic levels, organized around clusters of interactional properties. These are not properties that are intrinsic to the objects but reflect our experience in somehow interacting with them. They are dependent on the human and interaction context (or an imaginary “human-like” actor, like when we envision ourselves being the software client of a software component. That is, we must put ourselves into the situation of an actor and interact with a prototypical member of a basic level entity. The properties on higher than basic levels are abstract and purpose or function-centric. Thus, purposes or goals must be made explicit to deal with “abstract” properties.

Some properties - the essential properties - are of central importance to our reasoning. Essential properties largely reflect our knowledge about the type characteristics of a thing (usually derived from a prototypical member). Thus, the properties which distinguish members of a category, naturally become second-class citizens.

Tenet V. Modeling of a concrete system preferably starts with modeling of basic level entities in a prototypical style. This leads to the representation of essential properties of the thing that is modeled. Non-essential properties must be discovered by asking the question of how members of the same type could differ.

Tenet VI. Functional properties allow an object to play a certain causal role in the interplay with other objects. Functional properties must be discovered through the analysis of “interplay” or processes. Note, the functional properties in software engineering are related to pure conceptual processes (information processing).

4.2.3 Properties - Conclusions

Based on the definitions made in the previous Sections of Chapter 4, we can summarize and conclude as follows:

- In principle, we perceive the concrete world around us in terms of entities that are either concrete components or concrete processes. We characterize these entities by means of properties.
- Individual concrete components have primary properties, whose values pertain to instances of components. All the other properties are secondary properties. They are the result of humans cognitively interacting with such components. In our works we apply this basic distinction not only in the context of humans interacting with components but also for components interacting with components. For instance, two artificial objects such as computers interacting with each other. Secondary properties of components therefore relate to their means of acting in a process.
- An orthogonal dimension to the primary/secondary dimension is the defining/distinguishing dimension, which we derived from the discourse on essential/non-essential properties. Defining (or essential) properties of an entity are those that the entity must instantiate to exist, i.e. they define the type the entity belongs to.

Besides our interpretations and hypotheses formulated in concrete tenets, which are intended to be used for a methodological support to understand and model socio-technical systems, we can depict the above bullets as illustrated in Figure 4-5.

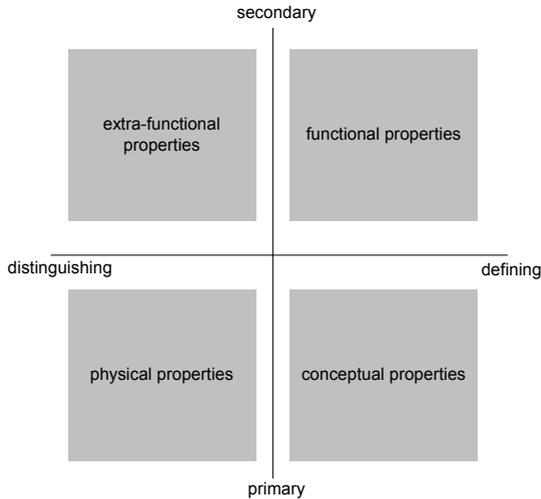


Figure 4-5: Concrete Components and their Properties

We mainly concentrated on two orthogonal dimensions of properties of concrete components, each dimension having two values (see Figure 4-5). Consequently, this yields four possible property types:

- Defining/secondary = functional properties: Properties that have a defining and secondary nature are determining the type of a component by means of its defined behavior, i.e. its response to stimuli. They are functional properties and relate to the functional role a component plays in a given matter/energy/information process. In terms of cognitive categories functional properties are basic level properties. *Providing cash* is an essential property of an ATM. As we shall see later in Section 4.3.1.1, functional properties express the essential information or matter/energy processing capabilities.
- Distinguishing/secondary = extra-functional properties: Being of secondary nature (i.e. behavior related) but distinguishing characterizes extra-functional properties. *Providing cash in a secure manner* is not essential for us to perceive the *Providing cash* functionality of an ATM. It merely is a distinguishing feature for the *Providing cash* type of service.
- Distinguishing/primary = physical properties: Values of properties that are primary and distinguishing pertain to individuals, i.e. instances of components. They are not essential to identifying the type of a component. They are essential to instances of types, however. Values of physical properties are frequently used to define the state of a concrete component. The *weight=62kg* is a physical property of a specific ATM.
- Defining/primary = conceptual properties: A property with a defining and primary nature is conceptual, i.e. we have promoted a physical property to define whether a thing is an instance of a conceptual (and thus non-basic) category. All ATMs *being more than 80kg* is such a conceptual property. It defines the conceptual category of things being heavier than 80kg. Note, being more than 80 kg is not essential to an ATM, but only to our conceptual category, which we might have defined for some purpose.

Figure 4-6 then illustrates our basic model of properties of concrete components as derived from the philosophical and cognitive schools of thought. All but the conceptual properties have in common that they are exhibited by the component and can therefore be observed by some empirical observation and measurement. Conceptual properties are normally not exhibited by a component as a result of some matter/energy/information stimuli. The conceptual properties, which we identified above as the result of mapping primary with defining properties, are only a small subset of the infinite number of possible conceptual properties. The only reason we brought them up was to be able to explain that the pair primary/defining yields a potentially meaningful combination, but as the word meaningful implies, one that is conceptual. Examples of further conceptual properties can be “made in Switzerland”, “developed by a CMM-level 3 certified company”, “business component”, “code test coverage 70%”, the *name* of a person, etc. They all have in common that while they are important they cannot be discovered by empirically observing the component. They all relate the thing in question to some other context. For example, “made in Switzerland” relates the thing to an information scheme that helps to identify the origin of manufacturing, “developed by a CMM-level 3 certified company” relates the thing to a certification scheme for software development process maturity, the *name* of a person relates the person to a naming scheme. Frequently, conceptual properties relate to non-basic level categorization.

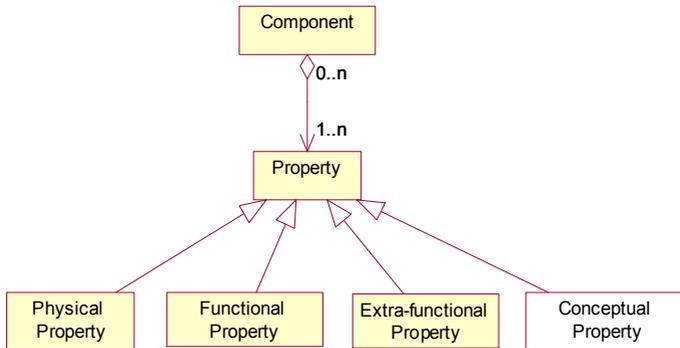


Figure 4-6: Conceptual Model of Properties of Concrete Components

4.3 Systems

Up to now we have discussed the notion of properties in the context of things or more specifically in the context of concrete or conceptual components mainly. The components of special interest to us, however, are systems. This Section shall therefore help to precisely yet generally define the notion of systems and to introduce the fundamental types of systems and their differences. In particular, the distinction between concrete and conceptual systems is important especially for understanding the fundamentally different types of non-functional properties in software engineering.

4.3.1 Concrete and Conceptual Systems

Because systems engineering is about understanding and thus modeling different types of systems, we wanted to understand whether there are commonalities and differences among types of systems that would justify a uniform or would justify a differentiated approach to modeling, respectively. These quests for a holistic understanding of systems lead us to systems science.

Systems science [70] has its origins in Van Bertalanffy's General System Theory [15]. This theory posits that certain characteristics are common to all types of systems. As Rosen [118] puts it, systems science looks at the systemhood of systems and not on the thinghood of them. While the systemhood is concerned with what is generic to any system, the thinghood of systems is determined by the adjective of a system (e.g. *physical* system, *biological* system, *axiomatic* system, *software* system, etc.). A system therefore exhibits properties that depend on its thinghood only, properties that depend on its systemhood only, and properties that depend on both. General System Theory is not only about systemhood but also about finding the demarcation line and relationships between the three.

The central concept in systems engineering is of course the notion of a system. Unfortunately, there are many definitions of systems in standard systems engineering literature: "A composite of components, which interact with one another, to exhibit observable properties and behaviors" [98], "An aggregation of end products and enabling products to achieve a given purpose" (EIA-632) [42], "An element, with structure, that exhibits observable properties and behaviors" [100]. These definitions reflect the level of generality that the authors of the referenced documents deemed appropriate. Using the words of Rosen, we would say they are to a lesser or greater extent biased by the thinghood the authors had in mind. However, they are not general enough to cater also for conceptual systems. Consequently, we subscribe to Miller's definition of system, which we found generic enough to be applicable in all circumstances, yet concrete enough to be useful:

"A system is a set of interacting units with relationships among them. The word "set" implies that the units have some common properties. These common properties are essential if the units are to interact or have relationships. The state of each unit is constrained by, conditioned by, or dependent on the state of other units. The units are coupled. Moreover, there is at least one measure of the sum of its units which is larger than the sum of that measure of its units." [88]

What constitutes a system is relative to an observer or set of observers. Miller used the notion of a *variable* to denote what it is that is of interest to an observer.

While there are many kinds of systems discussed in basic systems science, it seems agreed that there are two fundamentally different types of systems (Figure 4-7): *conceptual systems* and *concrete systems*. Both types are discussed separately in the subsequent sections. As a conclusive insight, we would summarize the interesting relationships between conceptual and concrete systems as depicted in Figure 4-7:

- While a conceptual system is always a designed system (i.e. created by man), a concrete system may be a designed or a natural system, or even a hybrid.
- A conceptual system may model ("conceptualize") zero or more concrete systems. Example: A UML Activity Chart - the conceptual system – may represent a particular business process within a company.
- A conceptual system may model ("conceptualize") zero or more conceptual systems. Example: A UML Sequence Diagram - one conceptual system – may represent the flow of method calls of an object-oriented Java program – another conceptual system.
- Every conceptual system must live in one or more concrete systems. Example: A UML Activity Chart - the conceptual system - must live in the mind of an interpreter, say a human - the concrete system. This is further discussed in Section 4.3.1.2.

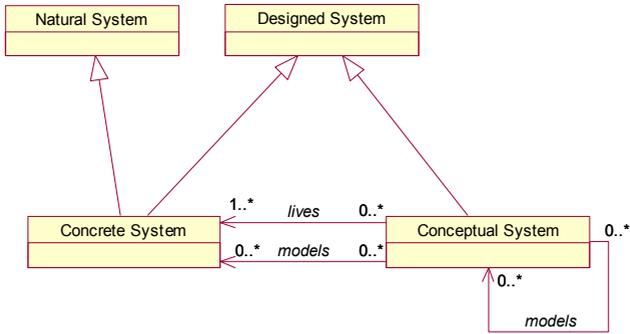


Figure 4-7: Basic Types of Systems

By reusing and extending our fundamental definition of entities in Figure 4-4, we can relate the concept of a system – conceptual and concrete – to the concept of a component as depicted in Figure 4-8.

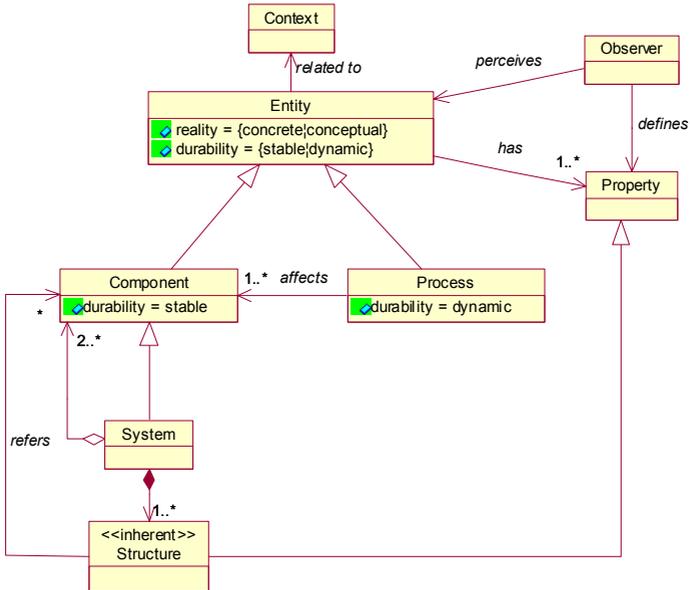


Figure 4-8: System as a Special Kind of Component

Any component whose inner working is of interest to us is called a system. It is in turn composed of components whose arrangement at a certain point in time is the system structure. Having a structure

is therefore an inherent property of a system, i.e. a systemhood property. We show the aggregation relationship from system to component as a UML aggregation rather than a composition aggregation because, in a conceptual system, the “part instance” (using UML jargon) can be included in more than one composite at a time. For example, the letter “X” is part of our natural language alphabet as well as of the Roman number system. The strong form of composition aggregation is often, but not always, appropriate to concrete systems. As an example for when this is not the case consider an individual person, who can be a component of a family and a component of a company at the same time.

4.3.1.1 Concrete Systems and Living Systems

Concrete systems are a non-random accumulation of matter and energy in physical time/space and their units and relationships are empirically determinable by some operation carried out by an observer. An organism, a human being, a social system, an electronic system (a radio or a computer) are examples of concrete systems. Table 8 defines the main concepts of the generic system definition given in Section 4.3.1 above when applied to concrete systems. The concrete systems of interest in the context of software-intensive systems are therefore computers, companies, user groups of software-intensive systems, project members of a development project, etc.

Table 8: Mapping of Miller’s Generic Systems Definition to Concrete Systems [88]

<i>Generic Concept</i>	<i>Meaning or Relevance in Concrete Systems</i>	<i>Example</i>
Unit	Components, parts, members. These units may also be concrete systems. Empirically determinable by some operation carried out by an observer.	Wheel, body, door, ...
Relationship	Can be of various sorts: spatial, temporal, spatiotemporal, and causal. Empirically determinable by some operation carried out by an observer.	Spatially and temporarily moving together
State	Its structure, represented by the set of values on some scale which its variables have at a given instant. This state always changes over time slowly or rapidly.	Wheel motion at the moment = 70 rpm
Observer	The observer of a concrete system distinguishes a concrete system from unorganized entities in its environment by (a) physical proximity, (b) similarity of units, (c) common fate of units, (d) recognizable pattern of units. The boundaries of concrete systems are discovered by empirical operations available to the general scientific community rather than set conceptually by a single observer.	Pedestrian (perceives car moving on a street)
Variable	Any property of a unit or relationship within a system which can be recognized by an observer who chooses to attend to it, which can potentially change over time, and whose change can potentially be measured by specific operations.	Wheel motion

We support the ideas of teleology¹² and assume that the behavior of concrete systems, such as physical, chemical, and biological systems, is purposeful. We also assume the correctness of Banathy’s view [10] that a key consideration in making distinctions among various types of concrete systems is the issue of how much freedom does the system have to select its purpose or goals, and how

¹² Teleology: The study of evidences of design in nature. [...] The use of design or purpose as an explanation of natural phenomena [84]

much freedom to select the methods and tools to achieve these goals. Based on these two assumptions we can infer that one important aspect of systemic action (which is one of the four main aspects of systems inquiry, as discussed in Section 1.4) is to investigate any system from a goals and means viewpoint. Because of an engineer’s focus on synthesizing an artificial object with desired properties, we take the goals and means viewpoint as being equal to an “outside specification”- and an “inside realization” view of a system, respectively. We use this analogy, because an engineer has usually several means to design the internals of a system to achieve the system goals and the engineer decides upon a certain alternative through consideration of trade-offs.

Tenet VII. Any system shall be investigated and thus modeled from a goals- and a means viewpoint.

The fundamental concepts for any concrete system, which processes some form of matter/energy, are informally depicted in Figure 4-9. The figure is our synthesis of the system introduction found in [10] with our understanding of information and matter/energy as discussed by Miller in [88]. We therefore also show Figure 4-9 in the informal notation like used in [10]. Figure 4-12, which is discussed later, will formalize the contents of Figure 4-9 in the form of a meta-model represented in UML Class notation. However, the essential message of Figure 4-9 is that the concrete systems of interest to us (and as such to systems engineering) process *matter, energy, and information*, collectively seen as the input they receive or the output they produce. Further, they are composed of interacting parts, which we call components. These components are in turn systems. However, we treat them as systems only if we are interested in their structure. Thus, in principle any system is a component of its suprasystem, if it determines through the interaction with other components the suprasystem’s behaviors.

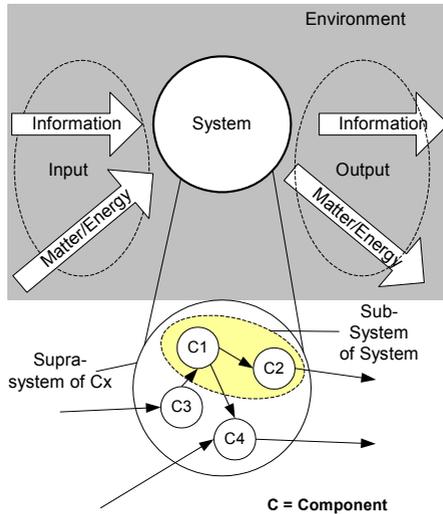


Figure 4-9: Informal Illustration of the Fundamental Concepts of a Concrete System

From this we conclude that we cannot model a system independent of considering/modeling (at least parts of) its suprasystem. If we are not doing it explicitly, it happens unconsciously. Software-intensive systems exist to serve, help or support people taking action in real world (as such usually called information systems) or supervise and control physical equipment and processes. Combinations

are of course possible. It is therefore a fundamental proposition “that in order to conceptualize, and so create, a system which serves, it is first necessary to conceptualize that which is served, since the way the latter is thought of will dictate what would be necessary to serve or support it” [28] p.10.

Tenet VIII. In order to model a concrete system we must have or create a model of its suprasystem.

Living Systems. In any taxonomy of systems, living systems are an important type of system. Business organizations, people, etc. are all special forms of living systems. Hence, especially in systems engineering, it should therefore be of central desire to have a scientifically accepted general model of living systems because IT-based systems are ultimately artifacts serving such living systems. Recall that according to Tenet VIII above, we need to model what is being served - the suprasystem - to conceptualize what is serving. To develop such a model of living systems, we investigated Miller’s Living Systems Theory [88].

In 1995 Miller published his second edition of a thorough cross-discipline analysis and synthesis of the functions and behavior of living systems. It is called Living Systems Theory (LST) and is a general theory about how all living systems “work” - from the individual biological cell to supranational organizations (such as the United Nations system)¹³. The goals of LST are to unify the scientific, often discipline-specific, approaches to study and model living systems. It applies the same general theory (i.e. it uses the same concepts and principles) recursively at all levels of complex living systems. LST represents a widely agreed conceptualization for different types of living systems. Miller’s unique contribution and thesis is the definition of seven levels of living systems with the reoccurring *same 19* types of subsystems for every type of system on any level. For us, however, the value of LST lies in the basic concepts and their relationships that we can adapt for our needs. Because the LST is a general theory, all concepts are metaphorical, i.e., they are meant to be algebraically translated to the particular living system in systemic inquiry. It is exactly this metaphorical value, which we take benefit from.

The central concept of LST is that of a **system**. Miller defines a system as cited in Section 4.3.1 above. By definition, living systems are open and thus constantly interact with their environment by means of information and matter-energy exchanges (cf. Figure 4-9).

The second most important concept is that of a **level**. According to Miller “...the universe contains a hierarchy of systems, each more advanced or ‘higher’ level made of systems of the lower levels”. He identifies seven distinct levels for living systems. Every living system belongs to one of seven defined levels and is composed of components that themselves are systems on the next lower level. It holds that these levels are practical (but not necessarily optimal) to describe the reality with enough simplicity. It is perfectly conceivable to define other levels. Miller chose these levels as they allowed him to do an extensive literature review related to these individual levels¹⁴, resulting in the suggestion on how systems on all these levels could be modeled using a common metamodel.

The most important principle is to view concrete systems from a structural and behavioral science perspective. This is in agreement with our fundamental assumption expressed in Figure 4-4: components and processes are the two basic entities we perceive in the world. According to the way we perceive living systems, Miller’s basic model of living systems introduces two principle perspectives of a living system:

- **“Physical” perspective (related to structural sciences):** A system is made of *components*. Components in turn are concrete systems, i.e. in the LST the term component always refers to *concrete* component. The higher-level system, of which a system is a component of, is called suprasystem. For example, an organ is the suprasystem of several cells. As mentioned above,

¹³ We will not go into Miller’s detailed definition of living systems because it is not essential to our discussion. The interested reader will find the characteristic criteria in [88] p.18.

¹⁴ Approx. 400 publications are analyzed for each level.

Miller defines seven levels of systems for living systems. These organizational levels are characterized by the fact that each level has the *same types of components but different specializations*. For example, on the level of organs, we find the brain, the heart, the liver, etc. all being composed of cells, which are components on the next lower level.

- **"Process" perspective (related to behavioral sciences):** A system has *subsystems*. Subsystems of a system are abstract concepts and always realized by the collaboration of one or more components (cf. also Figure 4-9). Subsystems interact by means of their components to achieve a desired overall matter/energy or information processing. Miller's central thesis is that every living system has the *same 19* critical subsystems (the list is given in Figure 4-10). They carry out the 19 essential processes of every living system. Miller defined eight subsystems that process matter and energy, nine subsystems that process information, and two subsystems that process matter, energy, and information. In general, subsystems serve to conceptualize reoccurring behavior. For example, two critical, reoccurring processes are those of storing matter/energy and memorizing information. Hence, Miller defined a *matter/energy storage* subsystem as well as a *memory subsystem* to be present in every living system. In an organization, as one example of a living system, a matter/energy storage subsystem could be realized by components of type warehouse personnel; a memory subsystem could be realized by components of type filing department personnel. It is interesting to note that Miller's notion of a defined set of subsystems is close to what software architecture would call an architectural style [89] for living systems, i.e. a general model of a set of architectural components in standing relationships.

From the above definition of perspectives we infer that every system is a component, but not necessarily a subsystem of its suprasystem. A component will therefore normally have one physical (structural) name, a concrete component name, and one or several role names depending on the subsystems it is involved.

For our general model of concrete, living systems we adapt Miller's conceptualization and represent it as depicted in Figure 4-10¹⁵. We use Miller's basic dimensions but we do not draw upon the specificity of Miller's theory, which comes from the definition of defined types of subsystems. In our future discourse and our metamodel of general systems (Section 4.3.4), we will substitute the term subsystem by conceptual or logical component. We do so because the term subsystem is heavily overloaded and does not convey its intended connotation to *conceptual* entities. In Figure 4-10, we acknowledge the specific contributions of LST by inclusion of notes where appropriate.

¹⁵ At first sight, a potential contradictory relationship between Component and Concrete System compared with Figure 4-8 (where System "is-a" special Component) might be observed. This is not the case however, because of the different contexts of discourse and the resulting terminology chosen. Any component of interest to Miller is per default a system.

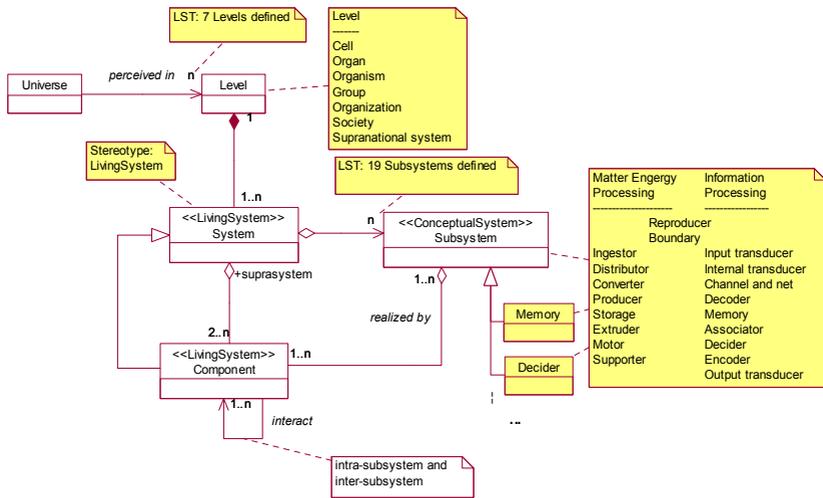


Figure 4-10: Conceptual Model of the Structural Concepts of a Concrete System (incl. LST References)

Besides the concepts of *system*, *suprasystem*, *level*, *component*, and *subsystem* (which, as mentioned above, we will call conceptual or logical component) we also use the following concepts adapted from LST:

1. *Space*: physical or geographical space (Euclidean space).
2. *Time*: the particular instant at which a *structure* exists or a *process* occurs or the measurable period over which a structure endures or a process continues.
3. *Matter and Energy*: Matter has mass and occupies physical space. Energy refers to the ability to do work.
4. *Information*: Patterned matter or energy, where the pattern encodes some meaning, i.e. an information concept is patterned energy or matter with a defined meaning. Hence, information is per definition a conceptual system, which relates units of information to units of meaning. A special form of information is the *template*, as defined in 8 below.
5. *Structure*: Structure of a system is the arrangement of its subsystems (process perspective) and its components (physical perspective) in three-dimensional space at a given time.
6. *Process*: All change over time of matter-energy or information in a system is process. Note that this definition of process also includes history. To express a process we need to be able to represent change. This is accomplished by showing the change of the value of variables that capture matter-energy or information at particular points in time, i.e. through changes of state-representing properties.
7. *Environment*: Immediate environment = suprasystem minus the System under Consideration (SuC). The entire environment includes the immediate environment plus all supra-suprasystems.

8. *Transmissions in concrete systems*: Takes place among subsystems (and thus components) within a system or among systems. Each transmission consists of some form of matter, some form of energy, or some form of information.
 - a. The *template*, genetic input or charter, of a system is the original information input that is the program for its later structure and process. It can be modified by later matter-energy or information inputs from its environment¹⁶.
 - b. *Information flows* are sequences of patterns over a channel in space-time from a transmitter to a receiver. If it is observed at system boundary we call it input or output.

4.3.1.2 Conceptual Systems

Conceptual systems may be purely logical or mathematical. Some sort of formal identity or isomorphism (“similarity or identity of form or shape or structure” [87]) with units and relationships of concrete systems may, but do not have to, exist. That is, we may have decided to model parts of a concrete system with a conceptual one (thus the association between them in Figure 4-7). However, all of the units (“concepts”) and relationships (“concept relationships”) of conceptual systems are selected/invented by “theorists”. This is in contrast to concrete systems where they are determined empirically by an observer. A theory, a language, a computer program, and any model and its representation are conceptual systems. Analogous to Table 8 for concrete systems, Table 9 maps the main concepts of the generic system definition given in Section 4.3.1 above to the specific ones in conceptual systems.

An important category of conceptual systems of interest in the context of software-intensive systems are in general subsumed under the term “software”, i.e. model representations such as machine code, or programming language source code, but also all the created model artifacts on the way to producing such source code. Simon would call these conceptual systems symbol systems [126]. Thus, OO programming languages or component-based programming constructs bring structure into the symbol system processed by computers in order to facilitate human processing. This is accomplished in that the representation of the behavior template (the program) is raised to a reasonably comprehensible level for humans.

Conceptual systems always live in one or more concrete systems, i.e. they are inexistent without their “instantiation” in concrete systems. For our purpose, we assume a computer program (e.g. represented as executable or source code) to be a conceptual system that encodes the behavior template. The latter lives in a computer (at execution time) and in humans (typically the programmers during development or code maintenance). This notion of instantiation allows us to conceptualize a programming language class (say a java class) or a software component not only as a conceptual but also as a concrete component. They are conceptual components in their source code form as part of the behavior template and they are concrete components in their instantiated form, i.e. when they are running as part of a program execution on a computer system.

Tenet IX. Any designed, concrete processing system has a behavior template, which is of the form of a conceptual system.

Tenet X. Every conceptual system lives in one or more concrete systems.

With respect to behavior template representation it is noteworthy that because humans do not function like computers we face the dichotomy between representations that are amenable for computer processing and representations that are amenable for human processing.

¹⁶ This notion of program was called “instruction” by von Neumann.

Table 9: Mapping of Miller’s Generic Systems Definition to Conceptual Systems [88]

<i>Generic Concept</i>	<i>Meaning or Relevance in Conceptual Systems</i>	<i>Example</i>
Unit	Terms: words (nouns, pronouns), numbers, other symbols (incl. those in computer programs).	John, 2
Relationship	A set of pairs of units, each pair being ordered in a similar way. Relationships are expressed by words (commonly verbs), or by logical or mathematical symbols, including those in computer programs, which represent operations (e.g. inclusion, exclusion, identity, addition, etc.)	Cubing = {(2,4), (3, 9), ...}; + = {(left operand, right operand), ...} Likes
State	The set of values on some scale, numerical or otherwise, which its variables have at a given instant.	
Variable	Each member of the set of units and relationships becomes a variable of the observer’s conceptual system. He may select variables from the set of units and relationships which exist in any concrete system or set of concrete systems, or he may select variables which have no connection to any concrete system. His conceptual system may be loose or precise, simple or elaborate.	Any of the examples given under Unit or Relationship
Observer	For his own purposes and on the basis of his own characteristics selects from an infinite number of units and relationships, particular sets to study.	John Likes

Opposed to concrete systems the notion of a variable is not defined through properties of units or of relationships because by definition properties are already concepts, i.e. inventions by humans, and thus units in a conceptual system.

It is important to mention that conceptual systems are organized knowledge or information in some form. Hence, what was discussed in Section 4.1 on modeling applies here: explicit sharing of conceptual models requires representations of them. And as a consequence, the properties of a conceptual system refer to

- properties of the representation bearer, which is a concrete component, e.g. a piece of paper or a diskette,
- properties related to the kind of representation, e.g. properties of a graphical representation such as how many graphical symbols are used, and
- properties related to the represented contents/meaning, e.g., graphically nested boxes means that the concept represented by the inner box is a physical part of the concept represented by the outer box.

Ultimately, all of these properties have an impact on how the representation can be interpreted/processed by concrete systems and whether the use of the conceptual system yields “desired” results.

4.3.1.3 Concrete and Conceptual Systems in Systems Engineering

In software-intensive systems engineering, three systems, including their possible suprasystems, are of immediate interest (see the UML Activity diagram of Figure 4-11, ignoring the dotted ellipses for now):

- (a) *The development group*, which is a concrete system (indicated by the left swim lane in Figure 4-11), employs a development method (a conceptual system) for a development project (a conceptual system). The development organization (not shown in the figure) is the immediate suprasystem of the development group.
- (b) *The software system*, which is a conceptual system (e.g. in the form of source code, or as a set of deployable/installable software artifacts), constitutes the output information of a development group and the input information (the “behavior template”) for a computer system.
- (c) *The computer system*, which is a concrete system installed at customer site, processes this initial input and any further information input to come. The initial input gives the computer system its new behavior (compared to none or its old one). The environment of the deployed computer system (e.g. the business organization, right swim lane) is the immediate suprasystem of the deployed computer system.

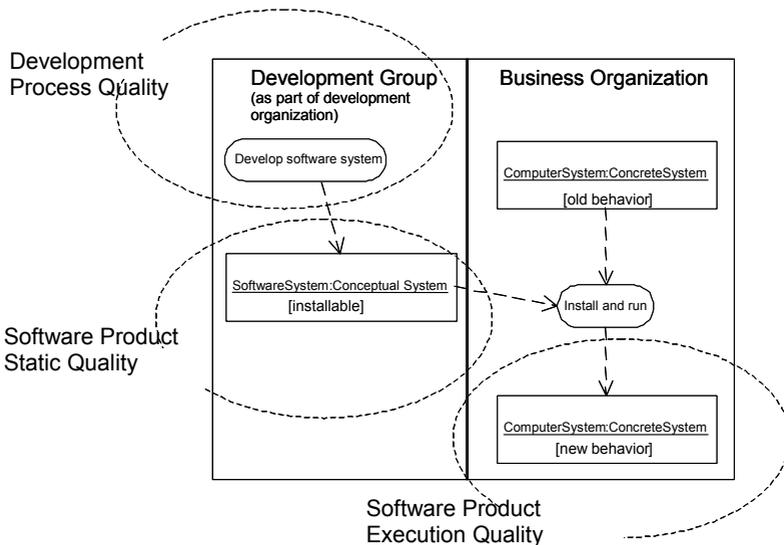


Figure 4-11: Informal Illustration of The Three Fundamental Types of Systems and their Relationship to Software Quality Domains

This realization is of interest insofar as that it illustrates the three distinct domains found in software quality (indicated through the dotted ellipses in Figure 4-11): process qualities, and product qualities divided into static product qualities and execution qualities:

- a) Process qualities; these qualities attempt to characterize the software development process and are typically found as criteria in the assessment of the process maturity. Examples are: configuration management usage, review strategies and implementation, documentation habits, etc. The underlying belief is that the process qualities have an impact on product qualities.
- b) Software product static qualities; these qualities attempt to characterize the static qualities of the software as a conceptual system (the symbol system), usually referring to the source code. The intent is to relate properties of source code components to design qualities such as coupling or encapsulation and ultimately to high-level quality characteristics such as reusability, complexity,

etc. Since we mostly treat the program (source) code as the relevant input, the software engineering community has developed fairly elaborate metrics to measure and describe static properties of source code [11]. However, since the systems are conceptual, the quality attributes are too. For instance, if the future of “programming” were in model-based execution of 2-D graphical models, we would have to redefine and probably invent new metrics for quality-carrying properties of such graphical models (see Section 8.2, Future Work).

- c) Software product execution qualities; these qualities attempt to characterize the deployed system at runtime. They represent the observable qualities of a software system in a concrete end-user context, i.e. they are essentially behavioral qualities and we can call them qualities of service. Examples are: reliability, security, availability, timeliness, etc.

Note that the pragmatics behind the quality model presented in the ISO 9126 (cf. Figure 3-4) validates our view in that it is also based on process quality, software product internal quality, and software product external quality attributes together with quality-in-use attributes. Hence, Section 4.3.1.3 can be understood to explain in a more general, or in a unified, way the philosophical underpinnings behind quality models in software engineering that are related to the ISO 1926 thinking.

4.3.2 A General Model for Concrete Systems

Concrete systems have two inherent and purely conceptual properties: *having structures*, and *being related to processes* (i.e. being involved in and therefore affected by processes). Because systems have fundamentally a conceptual or concrete reality (recall Figure 4-8), the structures are concrete (in LST this would have been the Component structure) or conceptual (in LST called the Subsystem structure). This causes the distinction between a decomposition of a system into conceptual- and concrete components, as discussed in Section 4.3.3. As a result of the composite nature of systems, processes may be internal or external depending on the observer’s viewpoint, i.e. they can be seen as occurring between systems or between components.

Figure 4-12 illustrates our general model for a Concrete System as is of interest to systems engineering. One could also call it an active system to denote a system, typically a socio-technical system, which is deployed in reality and dynamically provides response to stimuli. Figure 4-12 leverages on Figure 4-8. To make this explicit, we use no fill color for reused concepts in the class boxes of Figure 4-12. The latter is compatible also with our conceptualization of a living system (Figure 4-10). This compatibility is mentioned in the text below. The definitions and understanding of Figure 4-12 are listed in the subsequent bullet list¹⁷.

- A Concrete System of interest to us processes Components that are either or all of Matter, Energy, and Information. All these Components may be Systems themselves. The latter relationship is also expressed in Figure 4-10. If directed Information/Matter/Energy exchange or flow is observed at system boundary it is captured under the notion of Input and Output. Note that this definition is relative to the aggregation level of System and Component, i.e. there may be In- and Output occurrences related to constituent Components of a System, which are not visible occurrences at System boundary. While for systems engineering the processing (i.e. In- and Outputs) of type Matter, Energy, and Information are relevant, Information is usually sufficient for software(-only) engineering. Examples of processing are: a computer program (system) *verifies the grammar* (process) of a text document (system). Or, a truck (system) *transports* (process) containers of goods (systems).
- Concrete Systems of type living system or software-intensive system have a Behavior Template as the original information input as the starting point for their later behavior and structure. The Behavior Template can potentially be changed subsequently. We conceptualize

¹⁷ As a reader’s mnemonic aid, but not as a means to be able to trace details of the detailed discussion, Figure 4-13 depicts the models that were discussed so far and their overall relationships.

the Behavior Template as a conceptual system. Example: an executable program for a PC, an operation instruction for people.

- Components, Systems, Concrete Systems, In- and Output, and the Behavior Template have properties through the generalization/specialization relationships with Entity.
- The existence of the aggregation relationship between System and Component lead to the concept of *Structure* (in Figure 4-8), and it calls for use of levels to make explicit the relative positioning in potentially nested aggregations. Because System and Component may have concrete or conceptual reality (inherited from Entity), we can potentially have two kinds of structures, a concrete component structure and a conceptual component structure. In the LST-derived Figure 4-10, the respective structures are shown as the component- and subsystem hierarchy of concrete systems only.
- The occurrence of Input or Output is represented by the concept of a Process in Figure 4-8. This notion of Process is represented as the self-directed *interact* association of Component in the LST Figure 4-10. Input in any form represents the stimuli to which concrete systems respond in some form. The response can be visible only within the system or it can be visible at the system boundary. In the latter case it is Output of some form.

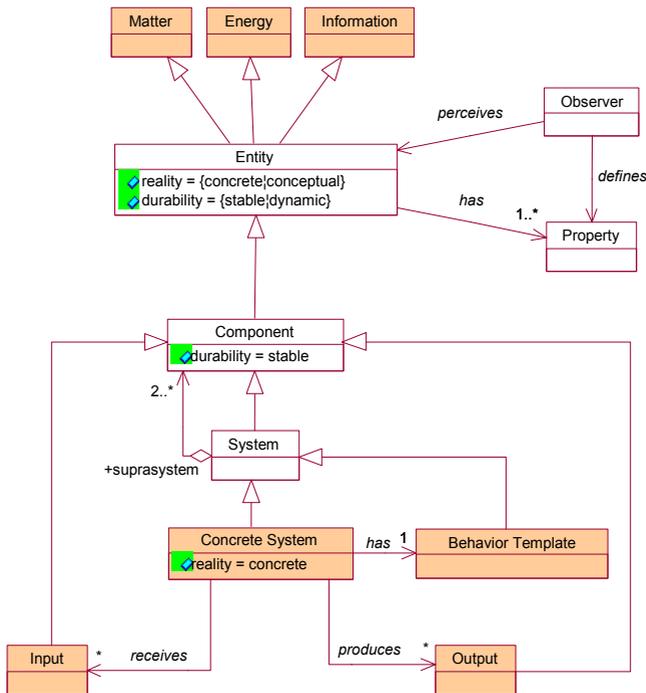


Figure 4-12: Conceptual Model of a Concrete System

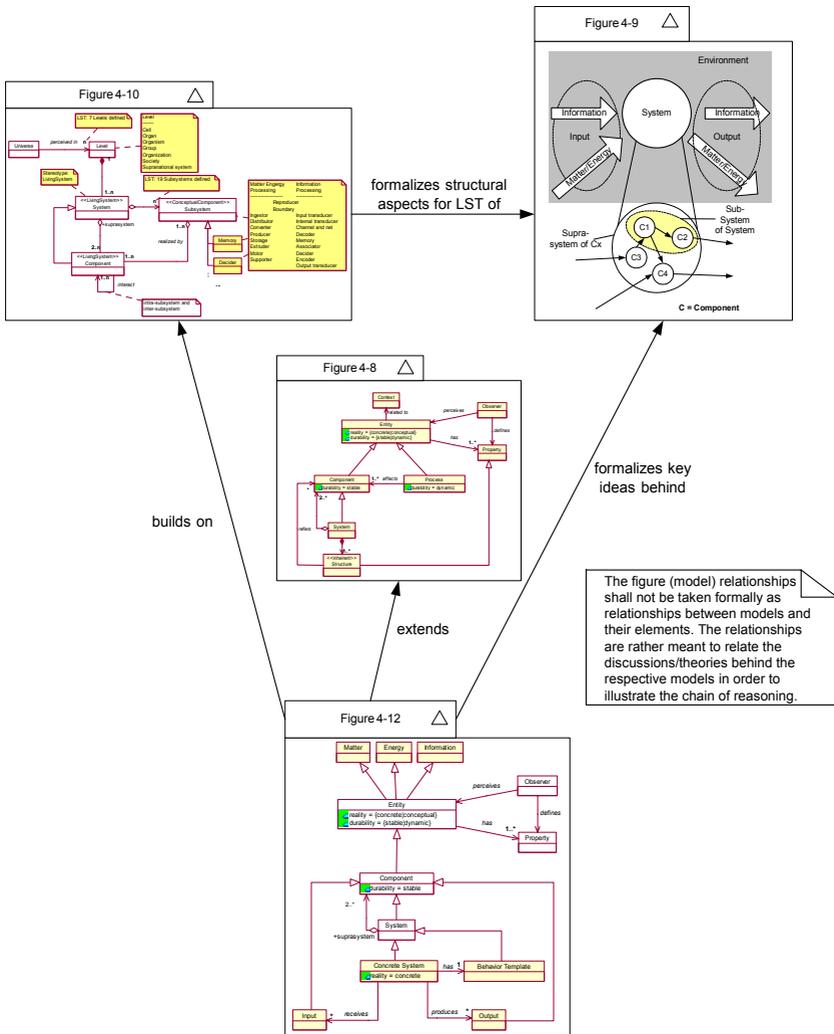


Figure 4-13: System Models in Section 4.3 and their Informal Relationships

4.3.3 Concrete versus Conceptual Decompositions

An important conclusion of our investigations of systems science is the explicit distinction between conceptual systems and concrete systems. As one of the important consequences for modeling systems, we introduce the notions of a concrete- (or physical) decomposition and a

conceptual (or logical) decomposition (Figure 4-14). This is a result not only of Miller's LST but also of the implications of the "is-a" relationship of System and Component in Figure 4-8 (and therefore also Figure 4-12).

The thesis of the LST is fundamentally grounded on the concept of a component hierarchy and the idea of a subsystem decomposition, which we could call a subsystem hierarchy. These two LST dimensions map onto our concrete decomposition hierarchy and conceptual decomposition hierarchy, respectively. This is depicted in Figure 4-14. While Miller essentially had two levels in the subsystem hierarchy¹⁸, we allow as many as needed to conceptually decompose a process so that its complexity is manageable by humans.

We arrive at the same two dimensions of decomposition if we analyze the implications of Figure 4-8 more thoroughly. Because a System is a special type of Component and because a Component can have concrete or conceptual reality, a System can be decomposed into conceptual or into concrete Components, which in turn yields a structure of concrete or conceptual Components. Hence, these two structures again represent our two dimensions.

The underlying definitions for this two-dimensional decomposition approach can be summarized as follows:

- Every component of further interest to us - because we want to analyze its inner workings or because we want to build a component artifact that shall exhibit defined properties - is a system that is composed of concrete and optionally conceptual components.
- The main reason to introduce conceptual components is to manage complexity. Hence, the goal is primarily to account for human comprehension. Conceptual components are logical units of behavior, and we therefore called them *Logical Components (LC)* in Figure 4-14. We use LC for abbreviation convenience, because concrete and conceptual component would both yield CC as abbreviation. However, boundaries of logical components are conceptual and they have some defined matter/energy/information processing. For simplicity of comprehension, humans typically use a small number of conceptual components rather than a large number of smaller-grained physical components to depict an operational model of the system by showing how these logical components collaborate to achieve an overall matter/energy/information processing. Examples of logical components typically dealt with in software engineering are in general architectural components defined for architectural styles such as client, server, pipe, filter, etc. In systems engineering also entities such as supplier, sales channel, etc. are logical components.
- Concrete components as opposed to logical components are those units that are deployed and recognized in physical reality, thus *Physical Components (PC)* in Figure 4-14. They are independent to the extent that they can be deployed into physical reality separately. As a metaphor to LST, they have their own, but of course contingent, living existence (in software: life-cycle). This starts with birth (instantiation or construction) and ends with death (destruction). PCs have their behavior template, and thus, they are behavioral units with some intelligence to respond to stimuli. In any system, where conceptual decompositions are used, physical components have typically a name that relates to their physical nature and one or several names that relate to their conceptual nature (i.e. roles in a process). For example, *Microsoft Internet Explorer* is the name of a physical component while its role maybe that of a *Web-Client* in a conceptual decomposition. Cognitively speaking, physical components are typically members of basic level categories, i.e. components with which we can have some form of physical interaction. Their boundaries are concrete. It is the physical components and their physical interaction that finally realize the overall matter/energy/information processing

¹⁸ For brevity, we only discussed one level in our report. This level was the one of the 19 reoccurring subsystems of living systems. Miller also showed that within some of these subsystems there is a reoccurring pattern of behavior that can again be conceptualized and generalized through "subsystems" of subsystems.

as conceptualized through logical components. Several physical components may realize one logical component, or one physical component may realize one or more conceptual components. For human comprehension a one to one mapping is of course preferable. The most important message is that **it is the physical components that are traded, reused, and based on which concrete systems are built. Hence, it is the properties of physical components that must be described as complete, as reliable, and as unambiguous as possible.**

An example illustration of this two-dimensional view of decomposition is given in Figure 4-14. A *System A* is decomposed into two collaborating logical components (LC A and LC B). They are in turn decomposed into further logical components. At some level, good enough for human reasoning, logical components are realized by physical components (e.g. LC BA by PC A3).

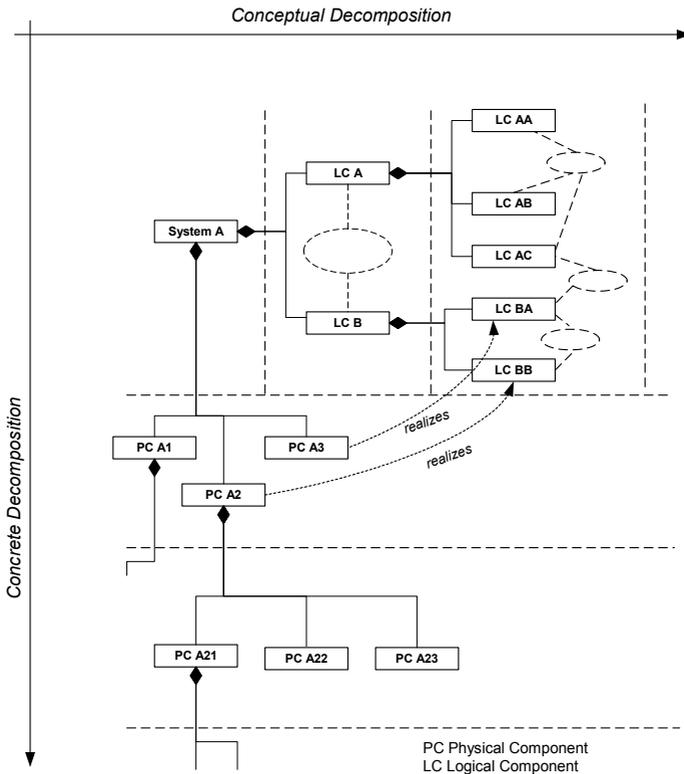


Figure 4-14: Conceptual vs. Physical Decomposition of a System

Based on this distinction between a conceptual and a concrete decomposition of systems into respective components, the following interesting observations can be made:

(a) The modeling approaches of some existing development methods consider one or the other dimension only, whereas others mix them completely. For example, UML Components does not

provide for a conceptual decomposition at all. Hence, it cannot deal with most architectural considerations. The ABD method defines a recursive hierarchic decomposition of conceptual components and stops when the leave nodes are to be realized by concrete components. The KobrA, as one of its basic principles, makes no difference between a conceptual and a concrete component hierarchy. The OOSEM mixes the two implicitly in that it uses logical and realization components but does not use the two in explicit hierarchies. The fundamental underpinning of OPM would allow for both types of hierarchies, but OPM does not use both hierarchies explicitly. None of the currently known methods to us actively and purposefully support a physical decomposition over more than one level. This is the kind of decomposition that is relevant to systems engineering. Especially in cases, where company organizations together with business processes and supporting IT and specific applications have to be modeled. The Systemic Enterprise Architecture Methodology (SEAM [142]), which is currently in development, is heavily focusing on the ideas of a physical decomposition as the main vehicle to partition the perceived reality and thus the model-based description of an enterprise and its components down to the level of programming language classes. In general, using both conceptual and concrete decompositions is primarily interesting for complicated systems. These are also the systems needing explicit architectural considerations [111].

(b) The explicit treatment of concrete components is important because it is the availability of published properties of concrete components that is relevant for system compositions based on preexisting components. It is the concrete components that are exchanged and potentially traded on the market and for which property descriptions are important.

(c) Through our hierarchical approach to modeling of systems, we automatically deal with an important issue in model-based requirements specification. Usually, in single level system specifications, it is very difficult to express non-functional requirements at the right level of abstraction. This is so because non-functional requirements at high-levels tend to be more informal [2]. For example: A user wants a distortion-free display of airplane positions for his air-traffic control system. This certainly translates into some form of update intervals required from positioning sensors. However, without having construction details of the control system we would not be able to allocate such an update rate to the right component or collaboration process. Hence, only a hierarchic decomposition approach automatically allows putting a property specification at its right place, i.e. allocating a property to the “right” system for the right stakeholder(s).

(d) The UML up to Version 1.50 does not provide for the language concepts to deal with either hierarchy. The recently released UML 2.0 provides for generic constructs to deal with hierarchies of components, but it needs to be verified (and could be considered possible future work) whether the defined semantics allows covering both types of hierarchies.

4.3.4 Systems - Conclusions

With the knowledge provided in Section 4.3, we can explain the transition from software (only) - to systems engineering in a more general way; namely that it amounts to a transition from the engineering of conceptual systems to the engineering of concrete systems. By this, we also say that it amounts to the transition from designing systems doing conceptual processing, i.e. information processing, to systems doing information/matter/energy processing. Software only, as a conceptual system, is just “brain-ware”, which has no natural engineering limits. It is physical world detached and as such similar to mathematics. Only when software is executed does it change from an ethereal entity to a tangible tool [137].

4.4 A Model of Systems *and* Properties

In this section we combine the property and system models, represented in the previous two Sections 4.2 and 4.3 and their respective figures. On a high level, we combine Figure 4-6 and Figure 4-12 into a conceptual model of systems *and* their properties.

To be able to focus the discussion on systems *and* properties, Figure 4-15 highlights the concepts and relationships that are of particular interest in that context. While Figure 4-15 is fully compatible with the previous figures (Figure 4-6, Figure 4-8, Figure 4-12) that buildup to Figure 4-15, it does not show all their details in order to not overload the figure and unnecessarily confuse the discussion¹⁹. Hence, all those relationships and specialized or generalized concepts of the previous figures that are not shown here are still valid. For example, in Figure 4-15 we have omitted the Input and Output entities that were shown in Figure 4-12 as special concepts to emphasize the matter/energy/information processing of concrete systems. To be more general, we show the concept of a Process as an explicit concept again. In the context of a Process we have Components *interacting*. This interaction affects the Components (*affects* was shown as an explicit association in Figure 4-8).

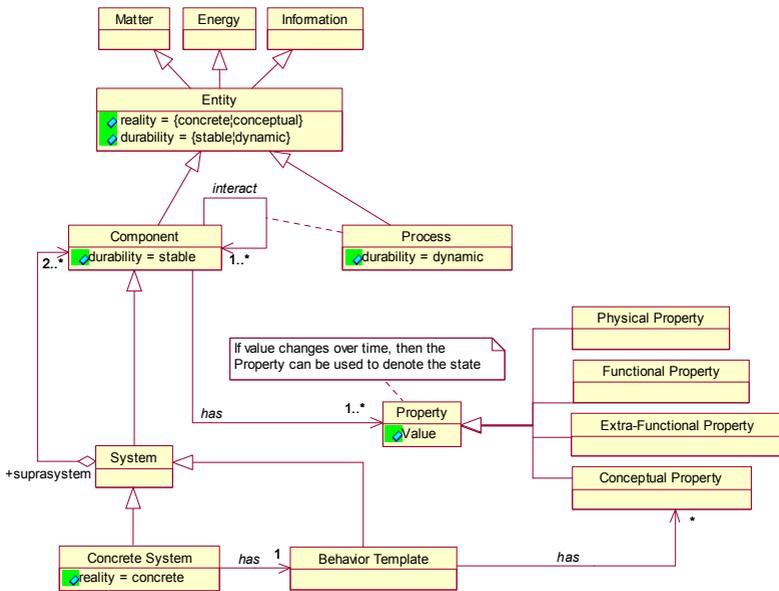


Figure 4-15: Conceptual Model of Concrete Systems and their Properties

The concepts of Figure 4-15 follow the definitions we made so far. Additionally, the following remarks and supplemental definitions apply:

- *Property value and Component state.* Every component has properties. Every property has a value on some scale. Every property can potentially be a state property. A state property is a

¹⁹ A complete synthesis of all the figures is provided as Figure C- 1 in Appendix C.

property which can be recognized by an observer who chooses to attend to it and whose value can potentially change over a time period in which the observer is interested. The state of a component is therefore defined through the set of values, which the attended state properties have at any given point in time. This comprehensive definition of state and state properties, which is represented in Figure 4-15, is very rich and allows expressing powerful phenomena or processes. For example, opposed to the typical use of state properties to express behavior of executing software, we can also view a functional property as a state property and then express the effect of a maintenance process (e.g. a security update) by declaring functionality change. As an example, see the OCL specification in the frame below. It checks that after an update a certain functionality, e.g. the functional property `openFile(..)` is removed and a new `openFileWithPerm(fname)` is defined.

```

context FileManager::securityUpdate(q819327)

let    oldMethodExists:Boolean = FileManager->exists (openFile(fname))

pre:  -- 'oldMethodExists' is not in the precondition because
        -- the update might run several times

post: FileManager->exists (openFileWithPerm(fname)) and
        not oldMethodExists
    
```

- *Behavior template on every level:* The typical development methods in software engineering do not pay special attention to the properties of behavior templates. In fact, they do not distinguish between different behavior templates for the different concrete systems. Of central focus to these methods (e.g. Kobra, UML Components, but also OOSEM) is only one behavior template, namely the implementation in programming language source code. Hence, the notion of behavior template per concrete system and level does not come up at all.
- *Behavior template and Conceptual Property.* We have already said that a Conceptual Property is used to capture the fact that any entity, and as such also a system and any of its constituents, can be part of an observer's arbitrary conceptual system and thus the property has an arbitrary definition dependent on that conceptual system. For us, one special class of conceptual properties is the one to describe characteristics of the behavior template. Because the behavior template is a conceptual system, we derive the properties largely based on its representation and our cognitive abilities to process that representation. The types of behavior template properties highly vary because they are dependent on the type of system and consequently on the type of behavior template. For example, the relevant behavior template for software systems is the deployed code. Properties are typically found in the realm of software metrics; e.g. static properties of the source code design (number of classes, number of methods per class, etc.). The behavior template for a human-activity system (such as a company) may take on the form of written instructions for a group of people (e.g. the development process guidelines, or the directives on how to process an enrollment application at a university). Consequently, we will have to consider different properties for such a behavior template.

5 Methodological Building Blocks Derived from Fundamentals

Objectives: Chapter 5 shall present how the body of knowledge provided in the previous Chapter can be combined and used to tackle the problems that were mentioned in the motivation of our thesis: identifying the relevant stakeholders, identifying the properties they care about in a given situation, providing support for property traceability, and a model-based treatment of the various types of properties.

After having read this chapter the reader will:

- know that for the task of identifying the relevant stakeholders, which is a prerequisite to elicit the relevant properties, we devised the general analysis model called “2-2-2”;
- understand how we propose to reason about properties and their relationships through the conceptual aid of property decomposition and traceability across hierarchies of systems;
- know how we propose to represent properties and property traceability in a model-integrated way based on the UML.

5.1 The 2-2-2 Model and its Use

In the previous Chapters we noticed the importance of the concepts *observer* and *context*. This importance appeared as early as in Section 4.2.1.1, in which we reasoned about what properties are good for. The importance was reflected also in the general models of systems (e.g. Figure 4-4 or Figure 4-8). We frequently substituted observer for stakeholder and we used context or environment depending on the discourse of discussion. The stakeholder is so important because it is him who defines what properties exist and what properties are to be ascribed to an entity (in our case to a system or a component). While the concrete suprasystem determines the immediate environment for a concrete system, the concrete system in which a conceptual system lives can be considered the environment of the conceptual system.

It follows that if we want to systematically identify properties we need to identify stakeholders first, which in turn are dependent on the environment of the system. To this end, we have developed the 2-2-2 model, which is a conceptual aid with which we can help to identify the breadth of properties in a principled manner. The principled manner refers to defining different perspectives on a system and identifying properties that are derived via the relevant stakeholders for these perspectives.

5.1.1 The 2-2-2 Model

The 2-2-2 model is a direct consequence of the following tenets, which we restate for convenience reasons:

- *Tenet VIII: In order to model a concrete system we must have or create a model of its suprasystem.*
- *Tenet IX: Any designed, concrete processing system has a behavior template, which is of the form of a conceptual system.*

- *Tenet X: Every conceptual system lives in one or more concrete systems.*
- *Tenet VII: Any system shall be investigated and thus modeled from a goals- and a means viewpoint.*

Our application of these tenets leads to basic assumptions that later form the orthogonal dimensions of the 2-2-2 model:

- (1) *Two systems (derived from Tenet VIII):* A (software) system in execution cannot be modeled without considering the system it is embedded in. Consequently, there are always at least two systems to be considered: the suprasystem and the system under consideration.
- (2) *Two viewpoints (derived from Tenet VII):* We must always explicitly analyze both the goals of a system and the means to achieve the goals.
- (3) *Life-cycle based domains of inquiry (Systemic action principle and Tenet X):* The life-cycle phases of the system engineering process (as defined for instance in [129] p.30) lend itself to define the relevant “domains of inquiry” as required by the systems science principle of systemic action [10]. This principle says that we shall apply any systems methodology in a defined functional context. By functional context, one refers to a process domain, i.e. context in which we want to apply the systems world-view, e.g. during the analysis of an existing system, during the design of a future system, during the implementation of a system, during the institutionalization (deployment) of a system, during systems operation, etc. We pick two main phases as points of reference - *operation and development* - although a finer grained partitioning would of course always be possible²⁰. The same choice of operation and development time can also be practically motivated by *Tenet IX*. The behavior template (a conceptual system) always lives in one or more concrete systems. In the case of software, these are the computer (the operational context) and the development group (the development context, where the behavior template and its associated artifacts are created).

The first two premises lead to what is depicted in Figure 5-1 (or one could say the “2-2”).

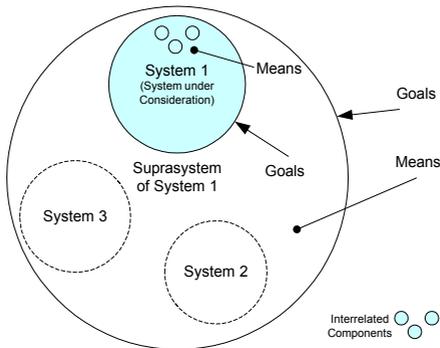


Figure 5-1: Two Relevant System Layers with “Goals-Means” Viewpoints

If we complement the “2-2” represented in Figure 5-1 with the third dimension, we obtain the 2-2-2 model as depicted in Figure 5-2. The third dimension represents discrete points in the life-cycle of a system. Each point stands for a defined life-cycle phase and as such represents the domain of inquiry, i.e. a limited domain of discourse in which we want to inquire/study our system. As

²⁰ In the quality construction model, discussed in Section 6.2, we are suggesting the purposeful use of a finer grained partition into life-cycle based phases.

mentioned above, we have chosen two distinct domains only. The usefulness of the inclusion of the third dimension was already observed in Section 4.3.1.3 in the context of systems and quality realms in software engineering. In fact, our 2-2-2 model can be viewed as an extension of Figure 4-11 (page 78). The extension amounts to an outside and inside view of the system under consideration.

In summary, Figure 5-2 represents a scheme of three dimensions with two reference points per dimension. Each of these eight combinations (or intersections in Figure 5-2) constitutes a meaningful perspective to study a system.

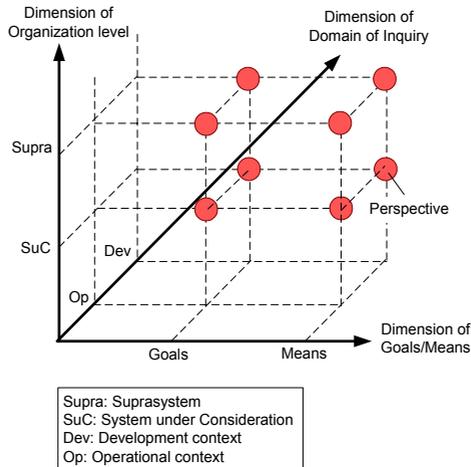


Figure 5-2: Informal Illustration of the 2-2-2 Model of Concrete Systems

The 2-2-2 model is the basis for a method for property identification. The study of each of its perspectives helps to discover properties specific to the perspective. The usage of the 2-2-2 model is captured in Figure 5-3:

1. Pick System under Consideration (SuC).
2. View SuC from the perspectives defined through the 2-2-2 principle: Each of the perspective corresponds to a special class of concerns that related stakeholders associate with a perspective.
3. Identify stakeholders per perspective: Every perspective is relevant for some stakeholders. To stay manageable, all the identified stakeholders can of course be grouped into categories of stakeholders or stakeholder roles (e.g. quality assurance staff, harvester role for reusable assets, etc.).
4. Identify stakeholder-specific properties. Since every perspective is relevant for certain types of stakeholders, which have specific concerns about the system, it follows that by investigating these concerns, we can discover to what properties of the system they lead. To this end, the “Goal-Question-Metric” (GQM [13]) approach can be used or scenario-based approaches in general (e.g. Quality Scenarios as defined in [9]). However, it is not essential to our idea which property elicitation approach is applied. This is the reason why we do not go into further details on the GQM or quality scenarios. We will revisit this topic when we indicate how we use the GQM in the context of a flexible quality model and its application to software components in Sections 6.2 and 7.1.3, respectively.

After the identification and collection of all required properties, the negotiation phase, as mentioned in Chapter 3, is next.

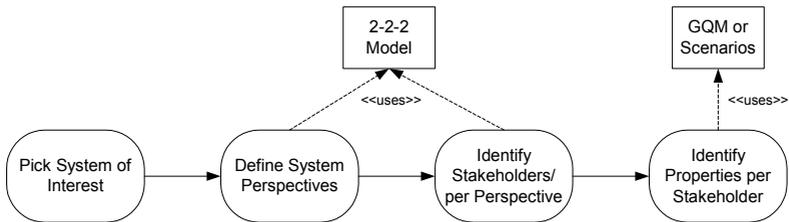


Figure 5-3: Sequence of Activities to Identifying Properties

The 2-2-2 model is in principle applicable to any system under consideration. The chosen one determines the nature of the perspectives and the types of stakeholders and therefore properties.

5.1.2 Stakeholder Discovery and Classification

We argued earlier that an important concept in describing and even identifying and gathering properties is that of a stakeholder. The most beneficial aspect and the actual practical contribution of the 2-2-2 principle is certainly the stakeholder discovery discussed in this Section.

There is consensus that the relevant properties of a system are dependent on the various stakeholders, who represent parties that have stakes on the behavior of the system or the way the system is being built. Ramesh [115] states that “high-end [requirements] traceability users” recognize stakeholder traceability as one of the most important aspects in their software process improvement programs. Also, almost all requirements templates/schemes proposed in literature carry an explicit reference to one or more stakeholders. Unfortunately, the identification and characterization of the relevant set of stakeholders for a certain system at a certain moment in time is largely unclear and thus done in an ad hoc way, supported only by experience and intuition. What one would prefer, instead, is to have some model that would guide the discovery and further classification of stakeholders. We believe that our 2-2-2 can provide exactly this. In particular, we believe that it is sufficient to consider two mutually dependent system layers and identify the “goals” and “means” stakeholders to be able to capture the entire set of relevant stakeholders for a system under consideration. It is important to mention that because we define hierarchies of systems, the relevant stakeholders are relative to the system under consideration. Consequently, the stakeholders for an IT-based business application (such as the one used in the example in Section 5.1.2.2) are not necessarily the same as the stakeholders for individual software components that are part of the application. For instance, if we consider a software component in its development context, we might discover that the roles and therefore stakeholders such as harvester, broker, etc., are key players [67]. Each might have certain specific demands, represented by required properties, on software components.

5.1.2.1 Generic Stakeholder Classification Framework

Here we propose a stakeholder classification scheme that is based on our 2-2-2 model. The life-cycle based domains constrain our universe of discourse, i.e. they support the separation of concerns. They represent snapshots, and thus portray the system under consideration at certain moments or for certain durations in time. Life-cycle based partitioning of domains of inquiry help to establish system boundaries and context, which limits the potential stakeholders to be considered.

The classification scheme has a generic layout as shown in Table 10. Section 5.1.2.2 shows a concrete, simple application of the framework by using the two above-mentioned domains of inquiry.

Table 10: Generic Stakeholder Classification Layout

Domain of inquiry	Goal stakeholder for Suprasystem	Means stakeholder for Suprasystem
	Goal stakeholder for “System under Consideration” (SuC)	Means stakeholder for SuC

The informal definition of the generic stakeholders is the following:

- *Goal stakeholder for Suprasystem*; the type of stakeholder that is interested in the behavior of the suprasystem only. It does not even know that our system under consideration is part of the suprasystem and it is also not interested in how the suprasystem achieves its behavior.
- *Means stakeholder for Suprasystem*; the type of stakeholder that is interested in the way the suprasystem achieves its behavior, i.e. its structure. Hence, these stakeholders would typically be concerned about some (or all) of the inner systems and their interactions.
- *Goal stakeholder for System under Consideration*: the type of stakeholder that is interested in the behavior of the SuC. It does not know or care about the means by which this behavior is achieved.
- *Means stakeholder for SuC*: the type of stakeholder that is interested in the way the SuC achieves its behavior. I.e., such a stakeholder is interested in the internal structure of the SuC.

Note that these abstract stakeholder classes can be applied to any system. For instance, the system under consideration may also be an individual software component that is a part of the application – the suprasystem.

5.1.2.2 An Example Application of the Classification Framework

In the following we present a simple example to illustrate the generic classification framework. Let us therefore assume we are a company producing e-commerce applications for online supermarkets.

The domains of inquiry define the suprasystem and the SuC. Our selection of these systems is influenced by business value chain considerations. A value chain is a sequence of actions, each adding value by transforming its input to value-added output, which in turn constitutes (part of) the input for the next action. For instance, an engineering company procures basic components and integrates them into a marketable system. The value-adding activity is the actual integration process, which is the core value-generating (and possibly protected) asset of this company. A second company might now procure such a system and produce an added value by employing the system to provide a service to their customers. This generic scenario is representative also for our e-commerce application.

In the “development” domain of inquiry, we define the suprasystem as being the development company with the development project being the SuC. The latter is producing the added value in the form of the development artifacts, of which a subset is then input to the supermarket to help create added value in their system. The produced artifacts – the behavior template - are the manifestation of the development company’s perception of the future behavior of the supermarket suprasystem and the role the e-commerce application plays in it.

Theoretically, all stakeholders identified in Table 10 will constrain, directly or indirectly, the realization of the software system to be developed. Ideally, we would analyze and model all

stakeholder-specific perspectives of the system to derive “to-be-built” properties. Since this undertaking is almost impossible to do, it is easy to understand why current development projects consider a small subset of stakeholders and models only. Current development methods usually identify in the analysis phase only the stakeholders for the SuC in the operation domain of inquiry. It becomes evident that these primary stakeholders directly relate to the properties that are discernable at system runtime (performance, usability, etc.). However, one can notice that by investigating the remaining stakeholders, we are lead also to other, e.g. development-related, properties.

Table 11: Stakeholders of an e-Commerce Application for a Supermarket

<i>Domain of inquiry</i>	<i>Informal description of system</i>	<i>Goal stakeholder</i>	<i>Means stakeholder</i>
Development	Suprasystem: Development company	Company board; company shareholders; technology/tool provider	Company line management; employees
	SuC: Development project	Company marketing/sales; product line managers; other company products ²¹ ; QA/process staff	Project member (programmer, architect, etc.); project management; other company projects; QA/process staff; company maintenance/support crew
Operation	Suprasystem: Company running the e-commerce application	Shopper; Supermarket suppliers (goods, etc.); Supermarket board; Supermarket shareholders;	Supermarket company line management; employees
	SuC: E-commerce application executing in target environment	Supermarket system user (back office, warehouse workers); in-house system administrator; in-house data maintenance personnel; other supermarket computer systems and applications (e.g. ERP)	Supermarket IT department; IT manufacturer/products for e-commerce platforms; e-commerce application vendor's hot line and maintenance crew

5.1.2.3 Beneficiaries of a Stakeholder Discovery and Classification Model

In addition to our own use, we believe that our stakeholder discovery scheme greatly improves the applicability of those methods in software engineering that, for the method to function, rely on having the right (or a representative) set of stakeholders identified. Many of these methods, however, assume that this is the case without providing any guidance to do so.

Although the concept of a stakeholder is found in almost all areas of software engineering, it is pervasive in requirements engineering. The emphasis is frequently put either on viewpoint-based requirements modeling that uses stakeholders as viewpoint representatives (for a survey see [129]), or it is put on stakeholder requirements elicitation and negotiation [117] [20], where stakeholders are

²¹ Which could reuse produced artifacts

used to elicit requirements from, and where stakeholders are in principle negotiating with each other about the importance and prioritization of requirements. In general, both of these emphases implicitly assume that the relevant set of stakeholders was identified somehow.

All of the mentioned work below have in common that they start with the assumption that the system and the stakeholders are obvious, i.e. defined.

Sommerville and Sawyer [129] introduce *viewpoints* and the orthogonal concept of *concerns*. A viewpoint is used as a concept to cluster stakeholder- or stakeholder class specific requirements. Concerns are orthogonal to viewpoints. Concerns represent systemic or 'holistic' high-level properties with the software engineering's connotation of quality attributes (e.g. safety, or cost). Concerns can be broken down into sub-concerns and finally questions. The latter are used to investigate in each viewpoint whether the requirements of the viewpoint conflict with the concerns. While the viewpoint and concern concepts are defined in detail, the all important stakeholder-relationship is not explicit and especially the viewpoint discovery (and thus the stakeholder discovery) is missing.

The Theory-W based spiral model [20], which is an extension to the original spiral model of software development introduced by Boehm [19], makes the task of identifying stakeholders ("candidate constituents") and their win conditions explicit. For that purpose, it defines a number of appropriate tasks that precede each spiral. However, the method is not explicit or prescriptive with respect to finding the set of stakeholders.

Pohl et al. [107] introduce a scenario-based approach to capture contextual knowledge about a system. This knowledge is in turn used to arrive at information that is important for requirements engineering. Scenarios are a means to elicit, validate, and negotiate requirements by expressing different kinds of interactions between stakeholders and the system. The importance of the "system around the system" is clearly supported by the fact that the model of the contextual information corresponds to a suprasystem model. Pohl et al. classify the context into three different worlds: usage-, system-, and subject world. With respect to systems science, the latter two capture the information that is derived from the analysis of the suprasystem. Especially the subject world is concerned with information that can be attributed to stakeholders that do not directly interact with the system and represents business goals, organizational policies, etc. In addition to not being explicit with respect to discovering stakeholders, Pohl et al. also acknowledge the fact that some "worlds", such as the development world, is missing (as opposed to our approach explained in Section 5.1.2.1).

Finally, the relatively young discipline of software architecture uses the concept of stakeholders as a means to typify the audience interested in architectural concerns. Stakeholders are used to assess the quality of an architecture [22] as well as to make the "customers" of an architect more concrete to derive the various aspects an architect has to consider [14]. Related to software architecture is the software product line business and in particular generative programming approaches [36], where stakeholder identification is part of the feature modeling process, but treated only as an informal and usually optional attachment for predominantly functional features.

5.2 Decomposing and Tracing Properties

In this Section we show how we conceptualize in a principled manner the fact that a property ascribed to a software-intensive system at one level is the result of its constituents and their properties. Consequently, we need to investigate how the manifestation of physical, functional, extra-functional, and conceptual properties at system level *relates to* (is realized by) components, processes, and properties at the next lower. In a model-driven era such relationships are preferably expressed by the respective models that describe the system and its internals.

The ability to keep track of the relationships across models and their representations is commonly called *traceability*. Traceability entails identifying and documenting the derivation path of properties (upward) and the allocation/flowdown path (downward) in the realization hierarchy of systems on the basis of relationships between the relevant model elements. In our case, these elements are the ones we

identified in our conceptual model of systems and properties derived in Chapter 4: component, structure, process, and the four fundamental types of properties of components.

The motivation for property traceability roots in its ability to serve as a basis for carrying out several kinds of analysis, including:

- *Design derivation analysis.* It allows capturing the sequence of design decisions (though not necessarily the rational). Hence, what system-level properties have given rise to the need for a given realization element?
- *Impact analysis.* It provides the basis to reason about how changes to one property might affect others. This includes the identification of which system-level property is affected if elements of the realization change. It may also include the identification of dependencies of system-level properties through shared realization elements somewhere down the road (discussed in Section 5.2.3 below). The latter is especially important because designers usually only have partial knowledge of the models, partly because not all aspects of a system are modeled, partly because they are modeled by different people or/and at different points in time.
- *Coverage analysis.* It allows determining whether all required properties have been taken into account, i.e. have been realized. Also, we can determine whether for a certain (or all) existing realization element(s) there is an actual need.

5.2.1 Realization-Oriented Decomposition Versus Other Forms of Property Decompositions

Before we proceed to the patterns of realization-oriented decomposition/traceability, we want to clarify by means of Figure 5-4 how two other types of property decompositions are related to the realization-oriented decomposition: (1) *classification oriented* quality attribute decompositions, and (2) the *analysis-oriented* decomposition for non-functional requirements.

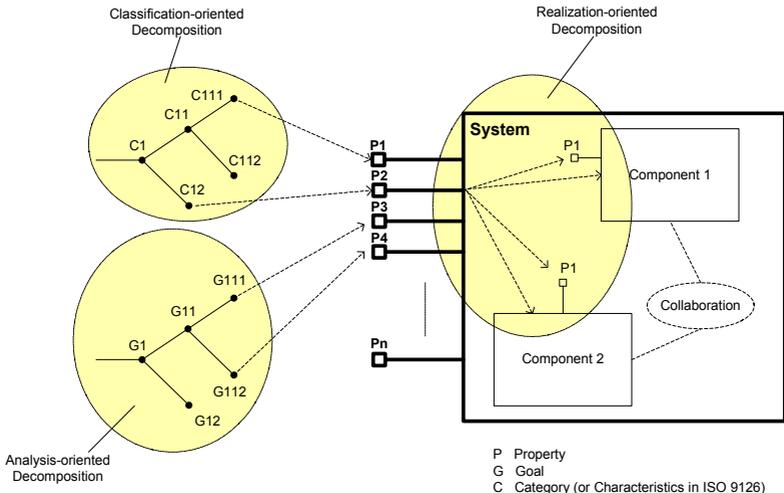


Figure 5-4: Different Types of Property Decompositions

Figure 5-4 shows a System and its ascribed properties $P_1 \dots P_n$. The System is composed of two components (Component 1 and Component 2) that engage in a Collaboration. In this simple example, every component has just one property P_1 . If we envision a designer who needs to design a System with the required properties $P_1 \dots P_n$, the constituents of the System would be called the realization elements.

In a *realization-oriented decomposition* we want to relate a system-level property to the elements that realize the system and that cause the property to manifest in the requested way. Figure 5-4 illustrates a simple case in which the Component 1 and 2 and their respective property P_1 realize the system-level property P_2 . Let us take the simple case where P_2 of the System expresses its *power consumption* in Watts. P_1 of Component 1 and 2 would simply be the respective consumption per component. Hence, P_2 of the System is no more than the sum of the two properties P_1 of the two components.

A *classification-oriented decomposition* on the contrary refers to a hierarchy of determinables and determinates, where the leaf determinates could be selected as the relevant, required properties of a system. Hence, it is a classification that serves the purpose of knowledge structuring. It represents a decomposition of high-level properties into more tangible ones so as to end with a set of quantifiable properties on some scale. The discussed ISO 9126 is a representative for such a classification because it defines a set of characteristics, which are decomposed into subcharacteristics, which in turn shall be decomposed into potentially measurable properties. Such a classification can therefore serve as starting point for defining the system-level properties to be realized. In Figure 5-4, such a classification is used to derive the required properties P_1 and P_2 of the system. For instance, P_1 could be the required physical property *power consumption*, whose value must be below a certain threshold. P_1 could have resulted from the ISO 9126 derived classification Efficiency (C1) -> Resource Utilization (C11) -> Power Consumption (C111).

An *analysis-oriented decomposition*, finally, refers to the decomposition of a requirement (often referred to as goal) to less ambiguous sub-goals and possible alternative means of satisfying them. Its main purpose is the support of trade-off and impact analysis. Its application is mainly found in requirements analysis [90]. In Figure 5-4, we assume that the system-level properties P_3 and P_4 are the result of an analysis-oriented decomposition. For example: the goal expressed in a requirement could have been to make the accounts of an ATM system secure (G1). We can understand this as the goal of reaching maximum confidentiality or/and maximum integrity (G11 and G12). For reaching maximum confidentiality we decide to have authorization based upon user identification (G111). This will become a required property (P_3) of the system.

Note that leaf nodes of a classification-oriented decomposition and an analysis-oriented decomposition may be the same.

5.2.2 Realization-Oriented Traceability Patterns

In the context of property traceability we are primarily interested in the realization-oriented decomposition, i.e. we assume that the relevant system-level properties, which need to be realized, have been identified somehow. Consequently, we want to be able to suggest patterns on how to realize and trace physical, functional, extra-functional, and conceptual properties at system level to specific structures, components, processes, and properties at the next lower level - the component level.

The principle foci of property realizations are presented in Table 12. By focus we mean that this is the primary or first concept of attention. From this primary concept the other dependent elements are derived. Concretely speaking:

- The focus for the realization of a physical property is Components and then their physical properties.
- The focus for the realization of a functional property is Process, from which then the involved Components and their functional properties are derived.

- The focus for the realization of an extra-functional property is Process or Structure, from which then Components and their functional- or still extra-functional properties follow. The latter refers to a case where the realization of an extra-functional property is deferred.
- The focus for the realization of a conceptual property strongly depends on the type of conceptual property. Hence, there is no clear focus. However, one category of conceptual properties of particular interest to software engineering are co-called architectural (or architecturally relevant) properties. So, we only show Structure as the main focus in Table 12 because the focus of architectural properties is Structure, from which then Components and Process follow.

This scheme shows the intricate situation when a functional property needs to be realized under consideration of an extra-functional property and potentially a conceptual property. This amounts to a realization trade-off considering at least two processes and a structural preference.

Table 12: Property Realization Foci

System- Level Property	Focus Concept		
	Process	Component	Structure
Physical Property		X	
Functional Property	X		
Extra-Functional Property	X		X
Conceptual Property			X

Note, we said that any property could be a state property, if it is an observed property whose value can presumably change over time. Consequently, we do not need a specific type of realization for state properties.

We will discuss every type of property type separately in the remaining paragraphs of this Section. Because we defined original methods for dealing with functional and some extra-functional properties, this Section features two subsections that will describe them respectively.

Physical Property

Principle of Property Realization: The value of a system level physical property is the result of the physical properties of its constituent concrete components. It follows an analytical property model (usually an engineering equation), which may depend on the configuration of the structural elements at a certain point in time. The analytical models are typically known in the respective domains.

Example: a) *Time to accelerate* a car given component weights, tire friction, power available at drive drain, etc. b) *Total weight* of the ATM as shipped to installation companies overseas, as the sum of the weights of all constituent concrete components.

Functional property

The system level property represents some response to stimuli of the environment. It therefore stands for a perceived or to be designed named form of matter/energy/information processing.

Typically, functional properties stand for system specific functionalities that are specifically engineered for. In fact, they were the reason why such a system was conceived in the first place. In

mature engineering disciplines these functionalities are realized based on the same solution patterns if the functionality is a known one. The reason why this is so, is that the overall solution can typically be proven to be optimal with respect to some criteria related to physical laws. In software engineering, however, which deals with information processing, functionality is frequently newly defined or the solution technologies or constraints are new. In addition, there are no hard-science like natural laws to define what is optimal processing of information. Hence, there is usually no existing “analytical model”.

Principle of Property Realization: The most general realization pattern is the mapping of the named system-level functional property (e.g. the operation name of a software component) to a named process in which components collaborate in a concerted interaction by means of their functional properties. The actual division of functionality and the kind of interaction is derived from both the required extra-functional and the required conceptual properties. Because we defined only a small number of concepts in our system and property model, we are able to suggest a general pattern for functional property realization, which we abbreviate with CCFP. CCFP stands for functional property is realized by a Collaboration (C), Components (C), and their Functional Properties (FP). Collaborations, which are representations of process, become 1st class design entities. This is discussed more in-depth in Section 5.2.2.1.

Example: A banking system’s software realization of the functional property “depositMoney([in]account, [in]amount, [out]newBalance)”²² has probably thousands of realizations in terms of the possible number and types of interacting components, types of interaction scenarios, and consequently, type and number of functional properties of these components.

Extra-Functional Property

The system-level extra-functional property is superimposed on a functional property. As such, the extra-functional property is also perceived (or to be designed) named functionality predicated to a functional property. The same type of extra-functional property may of course be superimposed on several functional properties. For example, several functionalities of an ATM require authorization (advancing cash, showing the current balance of an account, etc.) while others (like inserting the card, or selecting the user interface language) do not require this extra-functional property.

Principle of Property Realization: The realization of an extra-functional property is some form of behavior synthesis of the realization of the functional property and the realization of the extra-functional one. Because a single extra-functional property can be predicated to several functional properties, it might be possible to realize the extra-functional property once and share this realization with several functional property realizations. Some standard or reusable patterns of extra-functional property realizations are known today. They typically follow a reification-based approach [111], i.e. the property realization is made concrete through dedicated components. This is why current software component runtime infrastructures frequently provide a service infrastructure, which is the reification to cope with some of these properties. For example, in [111] we showed that the COM+ transaction service is the realization of the performance related properties of coherency and integrity, or the queued components service the realization of fault-tolerance and thus dependability (see also Table 13 on page 100). We present a technique on how we approach the reification-based realization of extra-functional properties in Section 5.2.2.2 below.

Example: A banking system’s software realization of the “depositMoney(..)” must provide *confidentiality by means of authorization* as one security-related extra-functional property. This may translate into a dedicated authentication server (a component) and an associated process that embraces the functional property realization.

²² Note, a comprehensive functional property specification should at least indicate the effect of its matter/energy/information processing through some form of pre-and post condition.

Conceptual Property

The conceptual system-level property is the most difficult to propose a general realization pattern for, because it does normally not relate to observable properties of the concrete system in execution. A conceptual property is any invented property that makes sense in a theoretical framework (i.e. a conceptual system) defined by some stakeholder. As mentioned earlier, the conceptual system may be anything from loose, informal, and subjective over more rigorous to mathematically strictly defined. As a consequence, there is no single general analytical model of conceptual system-level properties. They decompose, if at all, into any combination of properties of constituents.

However, we had defined two important conceptual systems in our general model of systems and properties: (a) the behavior template (recall Figure 4-15), which we treat as one internal conceptual component, and (b) the subsystem structure that represents the system as a set of related behavioral logical components (Figure 4-14).

Principle of Property Realization: Based on the two types of conceptual systems, behavior template and subsystem structure, we can at least say that an important subset of system-level conceptual properties is realized by properties of the behavior template and by the chosen subsystem structure. For the behavior template, which is information with its representation, these properties relate to representation-related properties. For the subsystem structure, these properties specifically relate to type and number of logical components and their collaborations. This affects the concrete components too, because the subsystem structure finally determines the structure of the concrete components.

Example: Examples for behavior template properties are for instance: The software is realized by *520'012 lines of C++ code*; The application must be *interoperable with COM and VB clients*. The latter translates into the behavior template (say the DLL) property *“only VB data types used”*.

An example for the hypothesis of the realization of development-related system-level properties as subsystem structures can be given with modifiability or portability. Both are architecturally relevant properties. They are realized almost exclusively as structures of logical components. In other words, these system-level properties determine the division of functionality for a required matter/energy/information processing into logical (or architectural) components. For example, modifiability calls for modularization so that expected future changes affect as few components as possible. Portability almost always calls for a portability layer (a logical component). Hence, while functional properties are realized by a process of collaborating components, is it the architectural properties that determine what functionality is assigned to what logical and finally concrete component. In saying so, we posit that during the design of a software system we would not even be interested in looking how a functional property is realized if we did not have implicitly or explicitly a required conceptual property in mind. Typically, an implicitly required system-level property in software design is modifiability.

In order to document the diversity of conceptual properties, which are not related to the behavior template or subsystem structure, consider the following: The system has *passed certification gate 3*; All developed products whose *break-even* was *not achieved within 2 years*;

5.2.2.1 Tracing Functional Properties: Collaborations 1st Class

In order to trace functional properties in a manageable way, we propose to make collaborations explicit 1st class model elements, i.e. model elements that have their mandatory explicit existence, own names, and explicit relationships to other model elements. A collaboration maps to the concept of a process in our general systems model. The basic assumption is that a functional property of a system is realized by a set of collaborating constituent components. If the collaboration is made an explicit concept in the model, we manage that every functional property must trace to one or more collaborations only, instead of tracing to a set of components and their functional properties. If the traces were to be manually maintained, this would soon overburden a designer. The reference to the

components, their roles and their functional properties is indirectly but unambiguously established because a collaboration is realized by collaborating components, which play *the roles defined in the context of the collaboration*. Note that a role is just a term of convenience defined in the context of a collaboration. It stands for a defined set of behaviors that the component, which plays that role, shall exhibit. As such it relates to a distinguishable use of a component. From a component point of view this use is modeled by one or more functional properties.

The CCFP pattern (Collaboration, Component, Functional Property) can be traced through collaborations being 1st class. The pattern therefore allows a recursive technique of “connecting” models, in which the functional properties are tied to realizations. This supports the notion of patterned model connectors, an idea that came up very recently [82].

Our discussion above is illustrated in UML notation in Figure 5-5²³. A *System 1*, itself part of a suprasystem (not shown) realizes Role 1 in the context of Collaboration 1. *System 1* does so through its two functional properties *operation a* and *operation b*, which are part of interface *Iface1*. Note that we showed an interface (a set of semantically related operations) only to indicate that our approach works with and without interfaces. It is not essential to our idea. However, *operation a* is realized through a *Collaboration a*, which defines three roles (*Role a1* to *Role a3*). Each role is played (realized) by a separate component, although a certain component could realize several roles, too. The detailed foregoing of the collaboration is determined through an interaction scenario that brings about the set of functional properties (i.e. again operations) of *Component 1* to *Component 3*. An example in the context of an ATM system is given in Section 5.3.2.4, when the specific use of the UML is discussed.

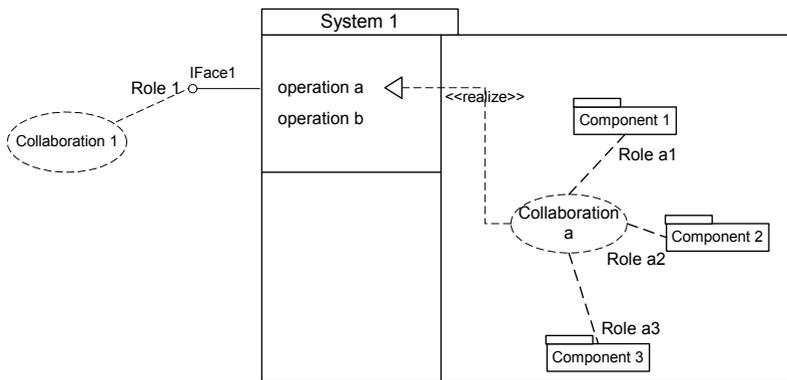


Figure 5-5: Collaboration-Based Realization of a Functional Property

5.2.2.2 Tracing Extra-Functional Properties: Extra-functional Roles

For the model-based realization of extra-functional properties we suggest an approach that introduces the notion of extra-functional roles. It is intended for extra-functional properties whose realizations amount to information processing as opposed to information bearer processing (recall the discussion in Section 4.2.2.1 or 4.2.2.4). The approach builds upon the importance of collaborations. This Section uses for its illustration generic concepts such as Component, Collaboration, Roles, etc.

²³ The figure uses standard notation for a UML Subsystem, its compartments, and the realization relationship. The latter type of relationship is defined only in the context of the subsystem notation. In Section 5.3.2, we will customize the UML to express our particular semantics. This will cause us not to use the subsystem realization relationship but a specialization of a UML Dependency instead.

Section 5.3.2.3, however, will take up the discussion again in the context of the UML usage by means of concrete examples.

In the context of software components or web-services the functional properties are usually called *services*. It is then also possible to call all extra-functional properties qualities of service (QoS). This is the term of convenience we use for our modeling approach.

Central to our idea is, firstly, the insight that extra-functional properties (aka. QoS) are secondary properties too, i.e. they amount to behavior and can be modeled in a role-based fashion. Secondly, we try to realize them whenever possible in a reification-based style. In this context reification means to represent key abstractions of a system as explicit objects (in this case related to the fulfillment of QoS requirements), which can then be reasoned about by the designer or even by the system through computational reflection. The rationale is to simplify the task of a designer by separating the design of the application into the steps (and even components) needed to realize functional aspects from those that address the extra-functional ones. Reification is the currently prevailing concept in the commercial software world. It is also the basic approach of the OMG to support QoS in their architecture [127]. In general, frameworks such as CORBA, COM+ or EJB provide application meta-services (reifications) and require some architectural support from their application components to address some of the commonly required QoS requirements (as an example see Table 13).

Table 13. COM+ Services and Their Support for Quality Requirements

QoS requirement on ...	COM+ Service
Performance (responsiveness, throughput)	In-memory database service; Object pooling
Performance (scalability, responsiveness) and dependability (fault-tolerance)	Dynamic load-balancing service
Dependability (fault-tolerance)	Queued components (message queuing)
Performance (integrity, coherency)	Transaction services
Security	Role based security services
Deployability (configurability) and usability (administrability)	Administration services

In the following we show that reification-based solutions can result from a systematic modeling and synthesis approach of functional and extra-functional roles.

Required system-level functional and extra-functional properties are the result of a system and its participation in processes with its environment. In software engineering, we have a one-sided look at these processes in that we are only interested in the roles that are taken over by the computer-based system to be built. From these roles we derive the high-level system operations. We know that any process in which the system and the environment are involved results in a system internal process. Talking in representation terms, a system's operation is realized by collaborations of computational objects, which we call conceptual components (logical units of behavior according to our definition in Section 4.3.3). We treat them as conceptual components because we have not yet committed to any concrete component that would realize them. As shown in Figure 5-6, in which the QoS ellipses should be ignored for now, the collaboration is composed of the roles *Rxx* and the interaction among roles. The roles are played by the components *CC1...CC3*. The interaction is inherent to the collaboration. A certain component may of course play different roles in different collaborations (such as *CC1* and *CC2* do). Besides fulfilling the functional property by means of a collaboration, the latter must also fulfill the required extra-functional property. For example, *Collaboration b* must also realize *QoS_b*. Because an extra-functional property does not manifest itself in isolation but only as a variation

of the manifestation of a functional property, we conclude that the realization of the combination functional/extra-functional property differs in some form from the realization of the functional property. What this means is that the collaborations differ in some form. Conceptually, we can approach this as a collaboration without extra-functional property and then “inject” the variation. Because we strive for a reification-based realization, we want to make the realization of the extra-functional property concrete in the form of explicit concepts. In the case of a collaboration, these concepts are limited to conceptual components, roles, and interaction.

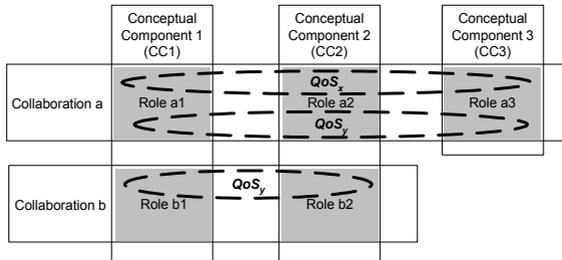


Figure 5-6: QoS_y as a Cross-Cutting Concern of Collaborations

In order to illustrate the idea of this kind of reification, we redraw the essence of Figure 5-6 and present it in Figure 5-7. It shows *Collaboration b* of Figure 5-6 again. In Figure 5-7(a), the required QoS property is represented as a constraint that the collaboration must satisfy.²⁴ The satisfaction of the constraint is conceptually realized through another collaboration among the same components (QoS_y collaboration in Figure 5-7(b)). This second collaboration is the root cause for new or modified properties of both conceptual components in that each of them now plays two roles: one to realize the functional aspect of the collaboration (Figure 5-7b, *Role b1* and *Role b2*), and another one to realize the extra-functional aspect of it (Figure 5-7b, QoS_yb1 and QoS_yb2). All this really says is that both components are to take over responsibilities in a process that consists of the joined collaborations. The assignment of concrete responsibilities to the conceptual components (“who does what”) is the next natural step in the refinement of the joined collaborations. This entails the decision of who is initializing the interaction and what the sequence of interactions looks like. In our example, this also requires that the QoS_y collaboration be broken down and responsibilities distributed among the components. But what component would be the natural place to put the bulk of the responsibility to? To make matters more complicated: based on Figure 5-6 we know that also our *Collaboration a* is required to meet QoS_y . Moreover, the fact that we have *CC3* involved in *Collaboration a*, which is not part of *Collaboration b*, could yield two different realization approaches for QoS_y . The resolution to this design question is to adhere to the rules of good design, i.e. design for change. This mandates, firstly, to encapsulate functionality in only a few (well encapsulated and loosely coupled) places, and secondly, if it needs to be distributed, to do it in such a way that later changes can be carried out in a uniform way. These rules lead us to the introduction of a new role, namely that of the “ QoS_y manager” (see Figure 5-8). It shall take over the bulk of the responsibility so that the other conceptual components are relieved to a hopefully uniform minimal responsibility.

²⁴ More recently (i.e. time-wise after our published idea in [111]), Wallnau et al. have proposed a very similar approach in the context of ensembles and their constraints ([136] p. 49 ff), whereby ensembles are the term of choice for collaborations of components.

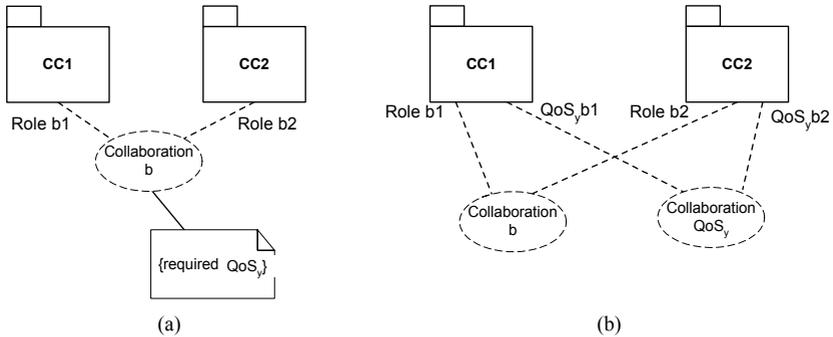


Figure 5-7: Collaboration b with the constraint to fulfill QoS_y (a) is broken into two collaborations (b)

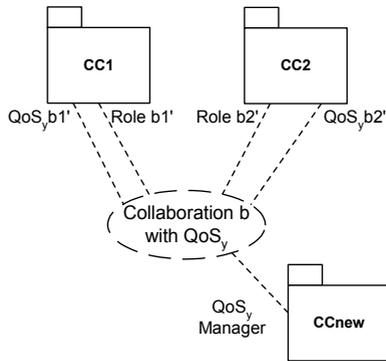


Figure 5-8: Distributing Responsibilities Yields New Roles

The synthesis of the functional with the extra-functional collaboration should thus be the basis for the realization of a system that meets both its required functional and extra-functional property. The principles by which roles and collaborations are synthesized shall follow the techniques suggested by various separation of concern approaches (e.g. [68] [86]). We therefore distinguish mainly three types of integration (or synthesis or weaving) options:

- Wrapping: The extra-functional collaboration executes partly before and partly after the functional collaboration. For example, locking/unlocking of critical resources.
- Adjacent: The extra-functional collaboration executes completely before or after the functional collaboration. For example, a component subscribes to object pooling (available for instance in COM+ [106]) at its first instantiation or even at configuration/deployment time.
- Overriding: Borrowing from aspect-oriented languages this describes the case when the extra-functional collaboration executes instead of the functional collaboration. For example, a functional collaboration may not take place because the security-related collaboration does not allow for it.

Packaging the realization (implementation) of functional and extra-functional roles of the conceptual components into single entities yields the concrete components. Packaging the realization of the “QoS manager” into a single entity yields a concrete component as a result of the extra-functional property. In a software component based system, for instance, the former concrete components would become the deployable software components. They would behave according to the specified functionality and support a well-specified QoS management. The set of “QoS managers” would typically be realized as another concrete component – the component runtime infrastructure.

5.2.3 Mutual Property Influences

Properties have mutual influences on each other in almost any non-trivial system. This is especially true for non-functional properties. For example, building security measures into the system slows down performance and has a negative influence on usability. The former can be a result of the additional computation that might be needed for encryption (one chosen means of a realization of security). Usability might be affected because users might now need to enter passwords or pin codes (another chosen means of realization). In fact, mutual dependencies are the one big reason why the “divide and conquer” principle does not work as nicely for non-functional properties as it does for functional ones.

While we cannot solve the inherent problem of mutual property influence we suggest a means of analytically detecting dependencies in the chosen realizations for our systems. By chosen realization we mean the set of concrete leave components and their interactions in the concrete decomposition hierarchy. We leverage on the ideas formulated in [41] for source code dependency analysis. In principle, we highlight the shared use of common elements in the realization of different properties, i.e. we identify the fact that traces overlap in the use of realization elements, which must be *concrete* components.

The basic inherent characteristics of a trace are:

- Transitivity: If property X traces to Y , and Y to Z , then X traces to Z , too.
- Bi-directionality. If property X traces to Y then Y must at least trace to X .

Based on these simple characteristics we can identify a shared use of common realization elements. Because of the two basic characteristics of a trace, it is possible to infer that the system-level properties X and Y trace to each other, as follows:

If some system-level property X traces to some realization elements X_R and another system-level property Y traces to realization elements Y_R , then properties X and Y are dependent if the realization elements overlap.

Through the recursive use of traces through hierarchies one might detect mutual influences that crawl into the design at later stages. This is usually the case when realizations are spread over several different model representations (say over different diagrams). Dependencies within a single model representation on a single diagram can be relatively easy to detect and do not need background reasoning automatisms.

5.3 UML as a Systems Modeling Language

In this Section we discuss how we use the UML to represent the concepts of our basic system and property model and of the discussed realization traces. Because every substantial model-based system description is based on a number of complementary representations, the next subsection will first present how we organize an overall system description.

We believe that this Section 5.3 can serve as a basis to compile partial input for a proposal for OMG’s request for proposal for a UML profile for systems engineering [96, 100].

5.3.1 Organization of a System Description

We introduced the basic terminology and concepts for modeling in Section 4.1. The ultimate goal in modeling is to have an externalized model, i.e. a model representation, of a phenomenon of common interest. This representation shall be suitable for communicating the knowledge about the phenomenon and for doing the intended reasoning. In our case, the phenomena of interest are hierarchies of systems and we want to reason about their properties. This Section defines how such a representation - a *System Description* - is organized. The latter term is simply a more convenient term than “a set of viewpoint-specific system model representations”, as would be appropriate given Figure 4-1 or Figure 4-2 in Section 4.1. Figure 5-9 shows the organization of a System Description in relation to the modeling concepts we discussed in Section 4.1 (cf. Figure 4-2 on page 57).

A System Description encompasses the set of Model Representations, partitioned into Views. Views are used to organize the representation of Models according to the relevant Stakeholders and their Viewpoints. Because we want to model concrete systems that are hierarchic by nature, we have consciously introduced a special View – the Hierarchical Level. One could say this is a “superview” because we are the “superobserver” or “supermodeler”. More practically speaking, we are the *architect* who cares about all the concerns of all stakeholders. The Hierarchical Level therefore serves as a means to express the agreed partitioning of the perceived reality into Hierarchical Levels also in the System Description.

Note, if we did not make a distinction between Model and Model Representation, but simply combined them under the term model, the axis “System Description-Hierarchical Level-View-Model” would correspond to the MDA model organization with our proposal for introducing an explicit concept of Hierarchical Level [142].

To give an example of such a system description let us again take the ATM. As an enterprise architect (and thus a systems engineer) we could be charged with a task to support the improvement of the IT infrastructure in a national banking corporation, in particular with respect to ATM usage. Together with all stakeholders we agreed that we want to model the as-is situation at four hierarchical levels: *banking cooperation* with all its branches, *branch* with people and IT systems (among them the ATM), *ATM* with all its physical components, and *application software* per ATM.

On every level we could decide to have a number of views to group cohesive collections of models. These views correspond to stakeholder viewpoints. For example, we could choose to address the testing concerns in a testing-related view and then model the system and its concepts related to testing. In our hypothetical example of the ATM, we choose only one view, which represents the corresponding system under consideration in its operational context. According to our basic approach to conceptualize systems, the models organized by this view would consist of a process model and a (structural) component model. The representations for these two models could be chosen to be, respectively, several representations of the UML to express behavior (activity charts, sequence diagrams, etc.) and several representations of the UML to express structure (class- or component diagrams). The models and their representations must conform to Agreed Conceptualization and Agreed Representations, respectively.

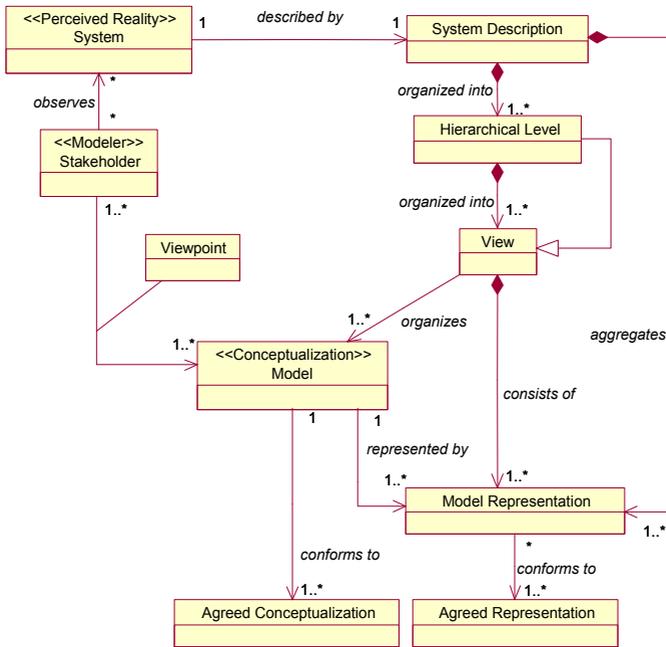


Figure 5-9: Conceptual Model of a System Description

5.3.2 UML Extension Proposal

Table 14 summarizes our proposal for the usage of the UML. It lists the defined concepts of our general system and property model and our choice of UML modeling concept(s). By UML modeling concept we mean the concepts defined in the UML semantics specification. The UML concepts are either taken as is or were defined based on the UML’s lightweight extension mechanisms. A practical constraint for our choice is that we would like to use current UML-based tool infrastructure, which is typically supporting UML V1.4, to the best extent possible. Lightweight extensions do not really prove adequate to deal with all of our requirements, i.e. some “semantic stretching” is needed. This complaint of inadequacy is supported through the fact that UML has introduced major changes in their recent UML 2.0. We have compiled the most important requirements from our systems modeling perspective in Section 8.2.1 and what concepts of the UML 2.0 are potentially fulfilling our requirements.

After Table 14, subsequent small Sections discuss the key extensions separately: Concrete Systems, Properties, and Property Traces.

Table 14: UML Profile for Modeling Hierarchical Systems and Properties

Entity to be represented	Standard UML / UML modified / Standard Extension / other	Comments
<i>Behavior Template</i>	See Conceptual System	
<i>Conceptual Property</i>	UML TaggedValue of type <<cp>> (see Section 5.3.2.2)	If there is an analytical model to calculate the value of a conceptual property, it shall be defined in mappingExpression.
<i>Conceptual System</i>	Any of UML Package, UML Component, or UML Artifact can be used to represent a behavior template. A UML Subsystem shall be used to represent a conceptual <i>behavioral</i> unit, i.e. a logical component as discussed in section 4.3.3.	In case of doubts, a stereotype <<behavior template>> or <<logical component>> can be used to more specifically hint at the intended use of the suggested UML elements.
<i>Concrete or Conceptual Component</i>	See Concrete System or Conceptual System. In addition, a stereotyped UML Class is possible.	
<i>Concrete System</i>	Stereotype <<Concrete System>> of UML Subsystem (see Section 5.3.2.1)	Uses a “C” in the fork symbol notation. This is a suboptimal choice. It would preferably be defined as a new abstract meta-class (see Section 5.3.2.1)
<i>Entity</i>	Stereotype of UML Class (or stereotype of any other subtype of Classifier)	
<i>Extra-functional Property</i>	UML TaggedValue of type <<efp>> (see Section 5.3.2.2)	Tagged value because UML Subsystem does not support StructuralFeature(s)
<i>Functional Property</i>	UML BehavioralFeature and its subtypes	
<i>Input / Output</i>	Stereotype of UML Class (or stereotype of any other subtype of Classifier, which also covers ConcreteSystem or ConceptualSystem)	Attributes of a Class shall be stereotyped with <<info>> for information or <<phys>> for physical matter input/output kinds
<i>Physical Property</i>	UML TaggedValue of type <<pp>> (see Section 5.3.2.2)	If there is an analytical model (e.g. a math. Equation) to calculate the value of a conceptual property, it shall be defined in mappingExpression.
<i>Process</i>	UML Collaboration	Collaboration representations (such as UML Patterns, sequence diagrams, etc.) may be complemented with other behavioral models.
<i>Property</i>	See different types of properties	Would preferably become a new meta-class similar to the existing abstract meta-class Feature

<i>Property Trace</i>	UML Abstraction and subtypes of Abstraction (see Section 5.3.2.3)	Abstraction is a subtype of Dependency and as such may bi-directionally relate to several suppliers/clients
<i>Stakeholder</i>	UML Actor	
Representation Management Concepts		
<i>View</i>	UML Model	
<i>Hierarchical Level</i>	UML Model	
<i>System Description</i>	UML Model	

5.3.2.1 Concrete System

In our case it is important that the modeling element representing a concrete system:

1. Can have behavior on its own, i.e. exceeding the provision of the individual behaviors of its constituent components;
2. Can be instantiable;
3. Can represent and refer to constituent components, which again can be concrete systems.

Possible candidates for stereotyping were Subsystem, Class, Component, and Package. Package and Component are ruled out because they clearly cannot satisfy criterion 1. Note that in the UML metamodel Component has the connotation of an implementation container, i.e. it does not have its own Features. Class is ruled out because of criterion 3. NestedClasses will not do because of its implementation focus and because a defined NestedClass may only be used by the declaring Class.

This leaves the concept of a Subsystem as the most useful candidate, although criterion 1 is not fully satisfied. In the UML metamodel, the Subsystem concept is a subclass of Classifier and Package. From the former it inherits the properties of having BehavioralFeatures (but constrained to be Operations and Receptions) and of *being instantiable*. From its being a Package it inherits its ability to denote any grouping of model elements. The new semantics of a Subsystem is its defined behavior specification with explicit relationships to the realization elements. This serves our purpose of having a goals and means view of a system (according to Tenet VII). However, a Subsystem was intended as a model management concept and its semantics is therefore biased from this original focus. For instance, the UML reference defines that “a subsystem has no behavior of its own” [96] p. 2-200. All behavior defined in the specification of a subsystem is jointly offered by the realization subset of the subsystem. But we may, depending on the interpretation of “jointly offered”, infer that a subsystem has emergent behavior.

For the time being and because of current tool environments we represent a ConcreteSystem as UML Subsystem with its graphical notation conventions but annotate the Subsystem fork icon with a C (see Figure 5-10). Using the Subsystem fork without a C indicates the conceptual system semantics.

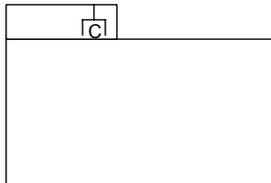


Figure 5-10: One possible Notation for a ConcreteSystem

All these shortcomings of modeling a system as a UML Subsystem would certainly justify an own “clean” definition for the concept of a System and particularly ConcreteSystem. We briefly present a possible suggestion in the rest of this Section.

A possible heavyweight extension for ConcreteSystem

If we had to live with the current CASE-tool supported version of UML (V1.4), we could envision extending the UML metamodel with a new meta-class called ConcreteSystem²⁵, which is defined as presented in Table 15. It would be a union of the semantics of the UML’s Classifier, Package, and Subsystem meta-classes with two exceptions: (1) the Subsystem well-formedness rule ([96] p. 2-196, rule [3]), which prevents StructuralFeatures, should be omitted, and (2) the Subsystem well-formedness rule [4], which defines possible references to containing elements, should be enhanced with ConcreteSystem.

Table 15: UML Meta-Model Extension for ConcreteSystem

<p><u>Description:</u></p> <p>A ConcreteSystem represents an arrangement of components in physical time/space. It can have system-level behavior that goes beyond provision of the behavior of its constituent components and it can have system-level structural (physical) properties whose values are depended on some or all of the structural (physical) properties of its constituent components.</p> <p>In the metamodel, ConcreteSystem is a child of Classifier and Package, having its own Features, and being associated with a set of constituent components, which again may be ConcreteSystems.</p>
<p><u>Inherited Features:</u></p> <p>ConcreteSystem inherits features as specified in Classifier and Package.</p>
<p><u>Stereotypes</u></p> <p><<person>> A person is a stereotyped ConcreteSystem that represents a human being and, as opposed to a technical system, would normally not need further decomposition.</p>
<p><u>Well-Formedness Rules</u></p> <p>[1] and [2] as UML Subsystem [3] A ConcreteSystem may only own or reference ConcreteSystems, Packages, Classes, DataTypes, Interfaces, UseCases, Actors, Subsystems, Signals, Associations, Generalizations, Dependencies, Constraints, Collaborations, StateMachines, and Stereotypes.</p> <pre> self.contents->forall (c c.oclIsKindOf(ConcreteSystem) or c.oclIsKindOf(Package) or ...) </pre>

Since the suitability of the UML 2.0 needs to be verified for representing and distinguishing concrete and conceptual systems, we do not further elaborate on possible meta-model extensions of UML 1.4.

²⁵ Actually, it would be nice to have the notion of a system and its two subtypes of concrete and conceptual system. The standard UML Subsystem would then nicely map to a conceptual system while concrete system has its (our) new semantics.

5.3.2.2 Properties

Because we chose to model a concrete system as a UML Subsystem, we select to model all but the functional properties as UML TaggedValues. This implies that we use textual rather than a separate visual notation.

The selection of TaggedValue is motivated by the factors listed below (for easier understanding, the relevant part of the UML metamodel is depicted in Figure 5-11):

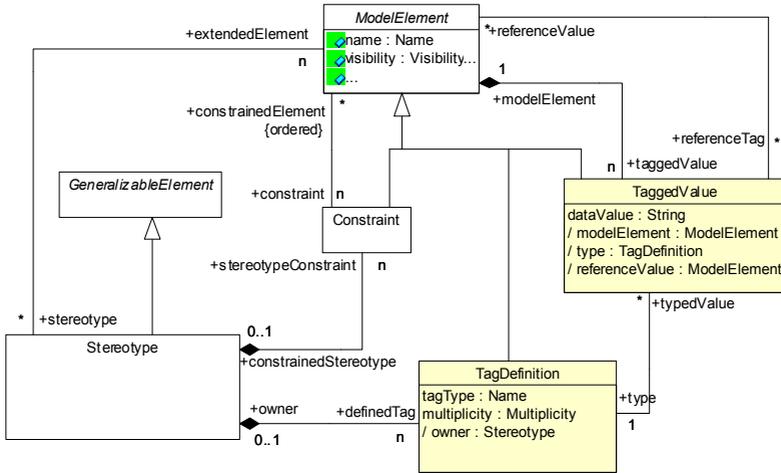


Figure 5-11: UML Metamodel for the Concept of a TaggedValue (Source [96])

- Most importantly, a UML Subsystem cannot have structural features. This rules out the direct use of Classifier Attributes, the other most promising choice.
- A TaggedValue is a keyword-value pair, where value is user-defined (i.e. modeler-defined) and can be encoded as a string of arbitrary complexity. The type of a TaggedValue can be defined via the concept of a TagDefinition. Hence, we can define tags that have structure so as to express the important information related to a property, such as description, metrics, related stakeholder, etc. Hence, an XML-based notation is possible to facilitate machine-assisted parsing of TaggedValues. This would allow defining a set of standardized properties, i.e. property templates, to harmonize the content and structure of domain-specific properties. Note that TagDefinitions are preferably defined together with stereotypes. We could therefore define our TaggedValues by TagDefinitions specifically aiming at our Subsystem stereotypes such as ConcreteSystem or even its further stereotypes that are potentially defined for different systems (ATMs, car, etc.).
- A TaggedValue is itself a ModelElement and we can therefore establish relationships between TaggedValues. This is needed for property traceability (see next Section 5.3.2.3).
- Instead of having a user-defined value, a TaggedValue may also reference another ModelElement directly. By this we can show that a property is in fact the direct result of values of other properties. Note however, that a user-defined value and a ModelElement reference may not be used together. That is, directly expressing a parameterized TaggedValue

(e.g. an equation or a mathematical function), where the parameter is another ModelElement, is not possible.

- The interpretation of TaggedValues is beyond the scope of the UML and must be determined by user or tool conventions. The UML authors assume that model editors provide help to search for defining or searching TaggedValues, but it is of course back-end tools that will evaluate tags.

5.3.2.3 Property Trace

The most straightforward way of tracing property relationships would be to simply establish a network of unqualified relationships (say “depends”) between the system-level property and its realization element(s), whatever the chosen realization elements are. This would require the developer to define and maintain all such relationships. But more importantly, because the semantics would be rather shallow, the tool support in the form of guidance and validation could only be minimal. The best we could say is that the client (system-level property) in some form depends on the supplier (realization elements and their properties) and that we would most likely have some form of impact if we changed one or the other. But the kind of dependency and impact is hardly discernable. For instance, we could not determine whether we would require all supplier properties for a client property, as we for instance do for physical properties. Because we want deeper kinds of traceability to perform richer analyses, at least taking into account our basic types of properties, we suggest a technique slightly more expressive than a simple “depends”.

To represent our semantically different property traces as defined in Section 5.2.2, we chose new classes of metamodel elements that are stereotypes of the defined UML concept of an *Abstraction*. Our choice is shown in its UML metamodel context in Figure 5-12 and further documented in Table 16.

Because all our representations of properties (taggedValue for physical, conceptual, and extra-functional properties, Operation for functional property) as well as our representations of realization elements (e.g. component, process, etc.) are subtypes of ModelElement, the concept of an Abstraction can be used to make bi-directional relationships relating several client and supplier ModelElements.

Every Abstraction has a *mapping* attribute that is of data type MappingExpression. The latter allows using the MappingExpression’s *language* attribute to give a reference to the kind of language used to denote the mapping and it allows by means of its *body* attribute to specify an arbitrary mapping between the client(s) and supplier(s) in the chosen language. Hence, we may choose natural language syntax or any domain specific language syntax (such as special mathematical language).

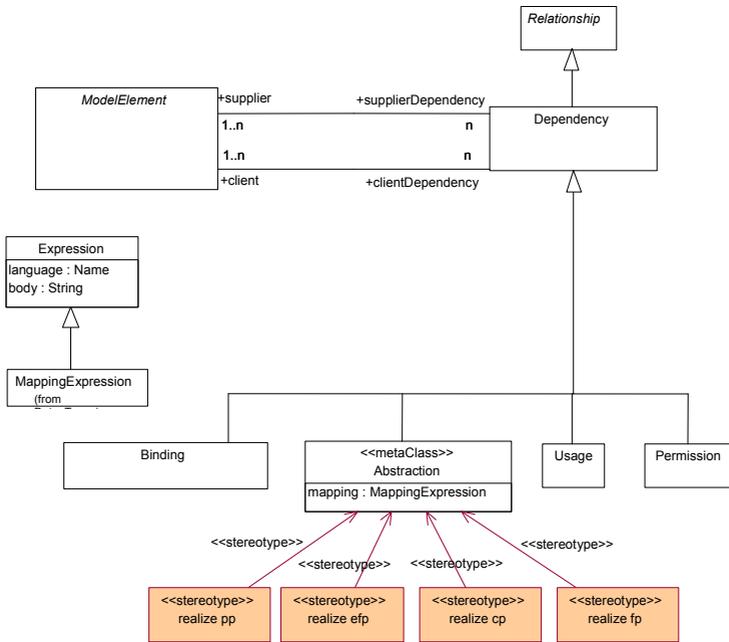


Figure 5-12: Property Trace Types as Stereotypes of UML Abstraction

Table 16: UML Stereotypes for Property Realizations

Type of Property Realization	Description	UML Stereotype
Realization of physical property	A dependency link exists between a system-level physical property and a set of lower level physical properties of structural elements, i.e. components. The elements in the set are combined to an expression (e.g. equation) that defines the system-level property.	<<realize pp>>
Realization of functional property	A dependency link exists between a Collaboration of components, i.e. the property link is established to the relevant behavior defining properties of the components involved in the realizing collaboration(s). We accomplish the trace through our CCFP pattern discussed in Section 5.2.2.1.	<<realize fp>>

<p>Realization of extra-functional</p>	<p>A dependency link exists between the extra-functional system-level property and its realizing Collaboration(s), with its components, their functional properties and possibly extra-functional properties. Here again, CCFP patterns are used and in addition extra-functional properties might be introduced for new functional properties of realizing components.</p>	<p><<realize efp>></p>
<p>Realization of conceptual property</p>	<p>We only foresee the special case where a system-level conceptual property traces to a set of lower level conceptual properties of structural elements.</p> <p>In the case where a system level conceptual property, say a developmental property, causes a certain overall division of functionality, the system-level property shall simply trace to the entire collection of realization elements, which therefore are best made explicit in a dedicated UML Package. The rationale behind the element selection can be placed into the MappingExpression of the single trace to the Package.</p>	<p><<realize cp>></p>

Note that realization deferral or combinations of deferral and realization are special cases of realization and as such taken care of by the latter. This is in principle applicable to all types of property realizations.

5.3.2.4 Examples of Properties and Trace Representations

In order to illustrate the use of the property and trace representations in UML, this Section gives a simple example of some aspects of an ATM.

We assume that we have to represent a partial model of an ATM. The ATM is part of a suprasystem. For the definition of the ATM's functional properties an ATM user (ATMUser), the ATM, and a bank back-office information system (BankIS) participate in a Collaboration (Figure 5-13).

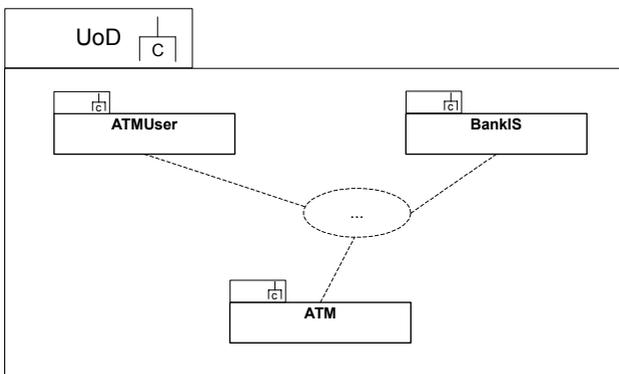


Figure 5-13: Collaborating Components in the Universe of Discourse

Among many other things, we assume that we are interested in the following properties:

- Physical property: *Weight*. The total weight is of interest for several stakeholders because of its implications on shipping means and charges, handling in the building construction, etc.
- Functional property: *showCurrentBalance()*. The ATM must provide the functionality to show the current balance of the account that is related to the ATM card.
- Extra-functional property: *Security*. Showing the current balance must be secure. In the example, this shall be realized by means of *authentication*.

Because we did not define a specific realization pattern for conceptual properties, we do also not elaborate on a specific UML-based trace example.

As a convention, which is normally to be enforced by graphical tools, we define that if TaggedValues are present only their tag name is shown to prevent cluttering of diagrams. Also, dotted lines before the curly end bracket indicates that more TaggedValues are present (similar to the Attribute notation in Class diagrams, e.g. see attributes of ModelElement in Figure 5-11). Further, trace names take on the name of the system-level property by default.

Physical property trace

Figure 5-14 shows the ATM depicted as a ConcreteSystem with its operation and realization compartments (left and right panes in the main rectangle). The physical property *weight* is depicted as a TaggedValue, which is realized by TaggedValues of the realizing structural components of the ATM. That is, ATM.weight traces to a set of physical properties of its structural elements, in this simple case also to the physical property *weight* of the realizing components. Because current tools do normally not allow showing the mapping attribute in diagrams (but instead show it in a separate window in the diagram editor or model reporting tools), we have added in this figure a UML Note indicating the MappingExpression of the <<realize pp>> dependency.

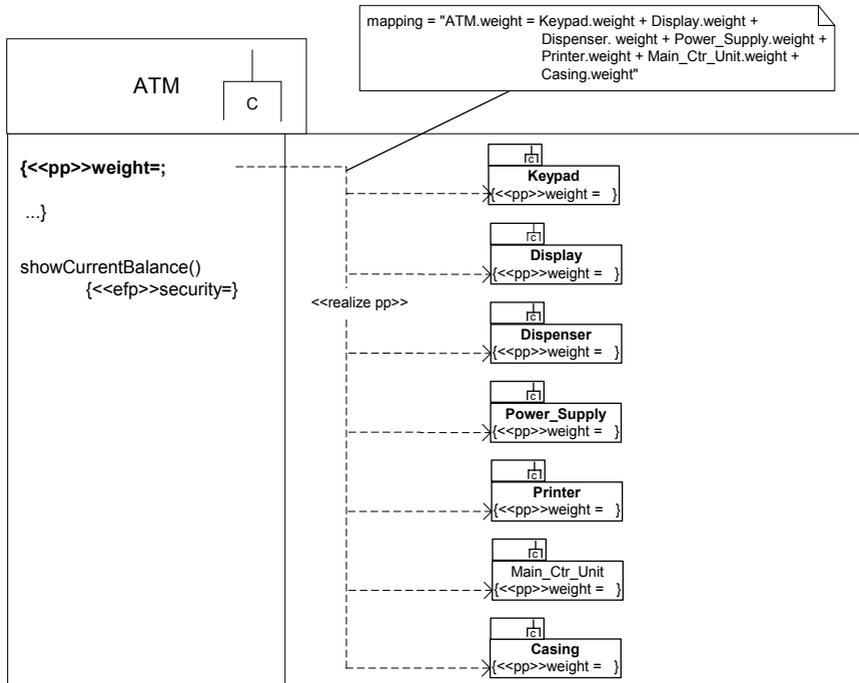


Figure 5-14: UML-Based Trace for a Physical System-Level Property

Functional property trace

The functional property “showCurrentBalance()” resulted from the analysis and design of the suprasystem collaborations. It is realized through a collaboration (CurrentBalanceDialog) of three roles played by components as depicted in the realization pane of the concrete system ATM (Figure 5-15). The components fulfill the roles relative to the collaboration. A refinement of the collaboration, for instance through a sequence diagram such as Figure 5-16 (other diagrams are possible too), shall help to determine the functional properties of the components. For instance, the Display component, playing the UserOutput role, would get the functional properties `displayCurrentBalance()` and `displayMainMenu()`, which are both available to interacting components. For simplicity reasons and to avoid indirections we assume UML Message-, UML Operation-, and UML Method names to be the same.

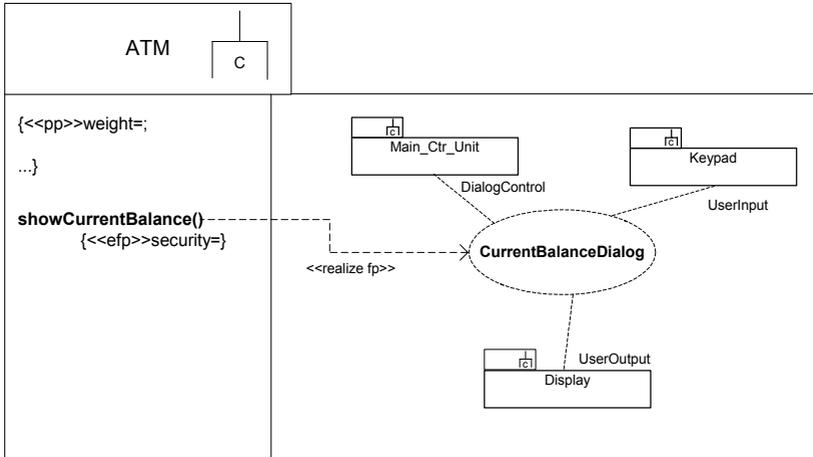


Figure 5-15: A Functional Property Tracing to its Realizing Collaboration

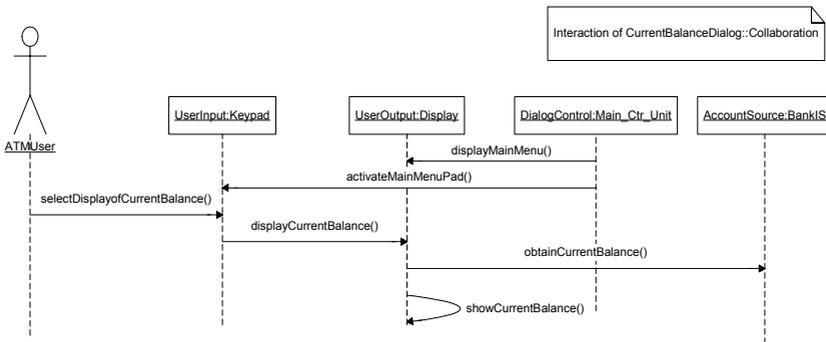


Figure 5-16: UML Sequence Diagram to Represent a Process and Determine the Functional Properties of Involved Components

Note that the sequence diagram is only for informative purposes to show one possible refinement, i.e. one possible allocation of functional properties to components. It shall not imply the “best” implementation design for the software. If we were already at software implementation design and if we had chosen a software component-based implementation, we would let the collaboration become one software component (the business logic concerned with the “currentBalanceDialog”) and the Keypad and Display become (proxy) software components, responsible mainly to encapsulate the keypad and display specific control of these devices. Consequently, a Keypad would probably never interact with a Display directly on a semantic level of “displayCurrentBalance()”. This would then be moderated by the business logic. Nevertheless, what we can see here is that if we chose the allocation of responsibilities as shown in Figure 5-16 and then pursued a further realization of the individual

components down to software, we might end up with a non-ideal software design with regards to maintainability and extendibility. It would be far from being optimal because if we wanted to change the logic of showing the current balance we have to understand and change software in at least three different components. From this little side-discussion we simply infer that the division of functionality may depend on further aspects such as the type of system and the type of its realizing components as well as on further (conceptual) properties to be considered. If we were at software component level for instance, we would employ a strategy with the rule: *Collaborations map to software components that shall contain the business logic*. This is possible and relatively easy because we made collaborations 1st class design artifacts.

Extra-functional property trace

Security is a high-level property and without further information too imprecise to convey what the stakeholder, for whom this property is relevant, had expected. We want an extra-functional property to be robust and expressive. What security could mean is that we want the account balance to be shown to the authorized people only, i.e. protect the legal card owner from someone else being able to easily obtain the balance information. This might in turn mean that we neither want someone else to easily use the card to display the balance on the ATM screen nor have someone else easily access the balance information source or spy on the transmission path. This is exactly a case of analysis-oriented decomposition. For brevity, let us assume that we realize cardholder authentication through the standard means of having a PIN code associated with the card.

According to the idea of extra-functional roles, the extra-functional property is realized by a collaboration realizing the extra-functional property (Authentication in Figure 5-17), which is superimposed on the collaboration realizing the functional property (CurrentBalanceDialog). The collaboration realizing the extra-functional property would typically involve at least the components that realize the functional property. In Figure 5-17, we called the combination of these two collaborations CurrentBalanceDialogWithAuthentication.

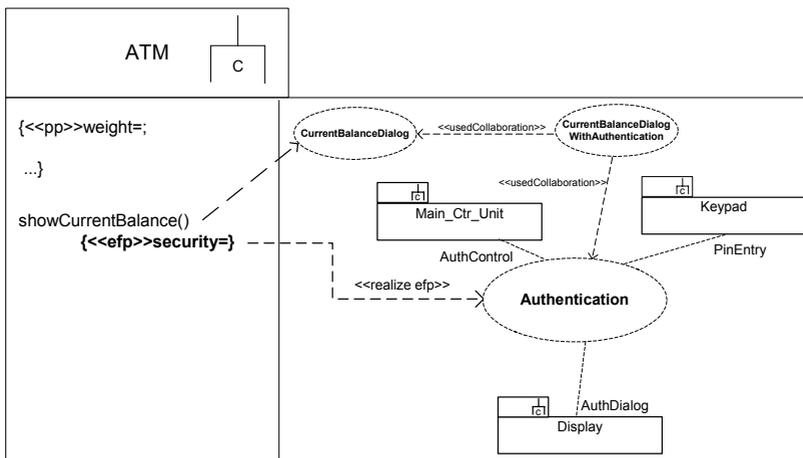


Figure 5-17: An Extra-Functional Property Tracing to its Realizing Collaboration

We are again using a sequence diagram (Figure 5-18, gray background) to determine the functional properties of the collaborating components that realize the Authentication collaboration. These properties are derived from the roles defined in the collaboration. For instance, we need a

component fulfilling the “Pin Entry” role in Figure 5-17. For this, we foresee the role-playing component to provide a functional property called “enterPinCode()”. The Keypad component was selected to take over this role. We also see that the Display component got more duties, which translated into more functional properties. In general, this is a good example to show that an extra-functional property at system level turns into functional properties of components realizing such an extra-functional property; in the words of software engineering, non-functional properties turn into functional ones.

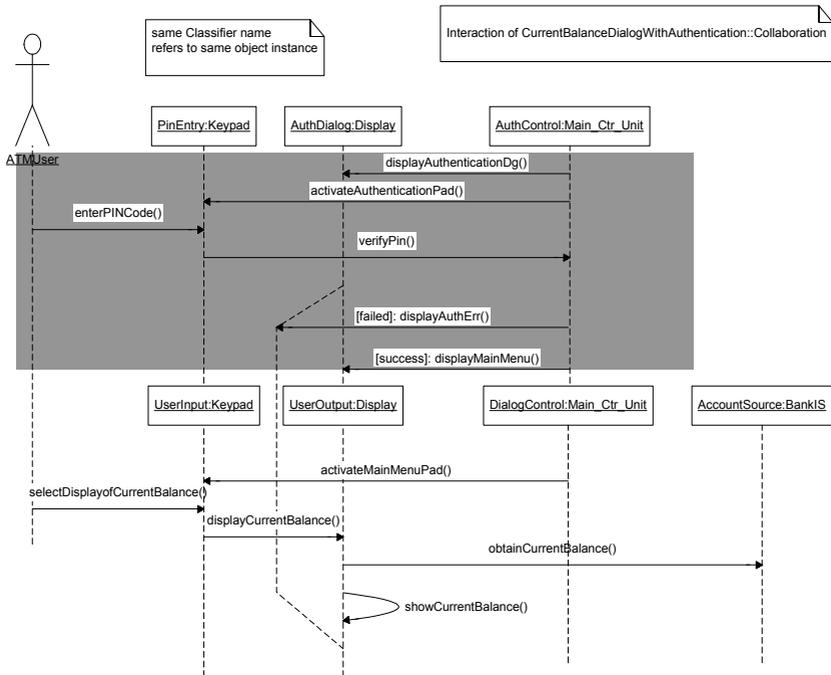


Figure 5-18: Refinement of the “Authentication” Collaboration and Integration with “CurrentBalanceDialog”

Figure 5-18 illustrates also the interaction that defines the “CurrentBalanceDialogWithAuthentication” as being an integration of the “Authentication”- with the “CurrentBalanceDialog” collaboration. The Authentication related interaction is indicated through the gray underlying color. The integration takes on the form of an *adjacent* integration in the case of authentication success and an *overriding* integration in the case of authentication failure. These two kinds of collaboration integrations were discussed in Section 5.2.2.2.

Note that the sequence diagram is not fully compliant with UML because sequence diagram integration is not foreseen directly. Hence, the alternative lifeline activation, which we use for the “failed authorization”, is not allowed to terminate at an instance lifeline of another UML ClassifierRole/object. We chose the representation only to make explicit the overriding integration.

However, we should mention that the recently released UML 2.0 provides for aggregate sequence diagrams, which would then allow for correctly expressing our needs.

In general, the same remark that was made above concerning software realization applies here: We would map the authentication collaboration to a dedicated software component. Referring to the discussion in Section 5.2.2.2, this software component would become what we called a QoS manager.

Summary of Part II

Part II presented our philosophy of modeling and our philosophy properties and systems. The philosophy concluded with a meta-model of systems and properties. This model essentially defined the notions of concrete and conceptual components and systems as well as of concrete and conceptual processes. It suggested the distinction of four basic types of properties of concrete components: physical properties, functional properties, extra-functional properties, and conceptual properties. If we consider the application of our philosophical underpinnings to be on two levels – generic methodological support and concrete methodological support in software engineering practice - then Part II discussed the first level. It presented general methodological building blocks to cope with properties in systems modeling. In particular, this encompassed the “2-2-2 model”, a method on how to discover and classify stakeholders for a given system and context, it encompassed a model for a principled traceability of the realization for our four basic types of properties, and it encompassed the representation of systems, properties, and property traces in the UML.

The subsequent Part III will elaborate on the second level of application of our basic philosophy in that concrete improvements for existing problems in software engineering are suggested.

Part III

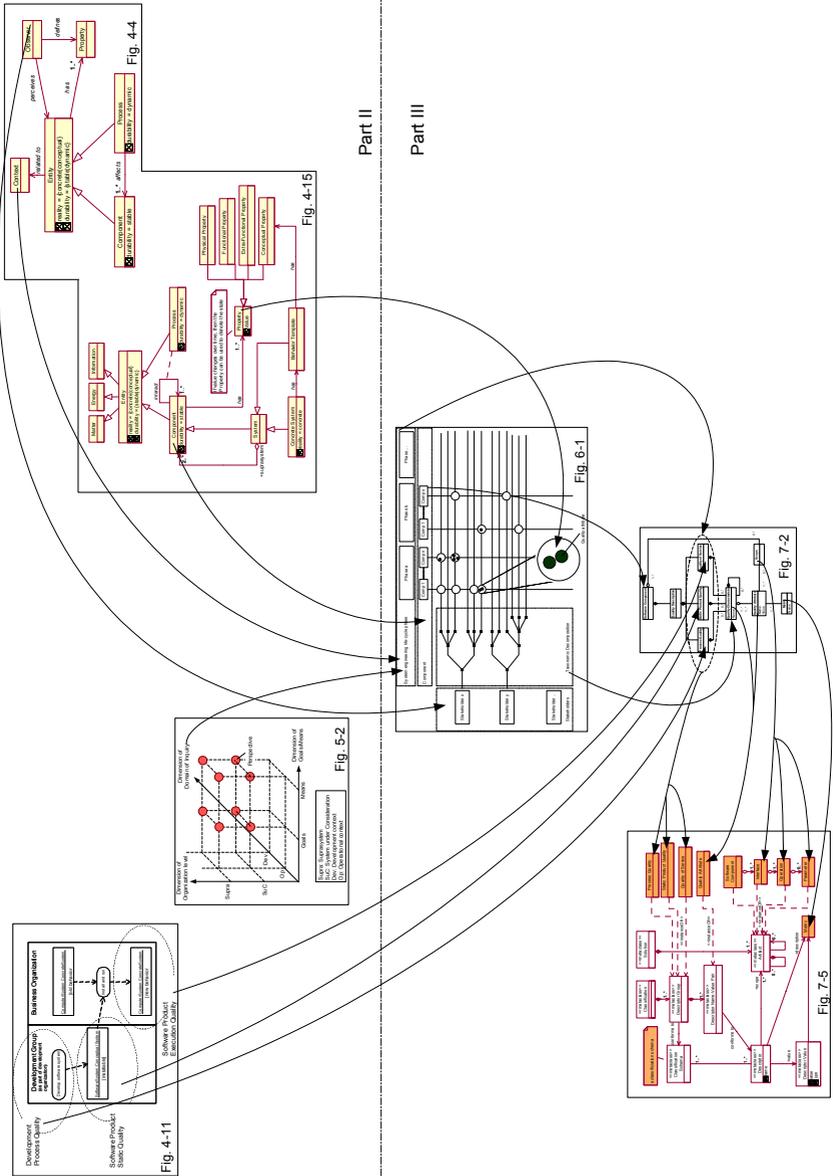
Application to Software Engineering

Part III presents possible concrete applications of our theoretical framework. They can have an immediate impact on software engineering practice. In fact, the reciprocal effect between our holistic understanding of properties and systems and our practical experience and challenges in component-based engineering and software reuse lead to the results presented in the Chapters of Part III. The results relate to the suggestion of a quality model construction framework and the description of properties of software components, the latter amounting to a component data sheet. With the quality model construction framework we acknowledge the fact that a single fixed quality model is not appropriate to approach today's variety of systems and components we want to ascribe properties to.

Part III further presents how the Reusable Asset Specification (RAS) can be used to represent a component data sheet in a standardized way and how we used the RAS for describing reusable assets in the ABB context.

As a mnemonic, the *Part III Overview Figure* presents the high-level relationships between the models and their concepts, which were derived in Part II, and the models and concepts, which will be derived in Part III. The figure is not intended to be studied in its entirety but its high-level relationships will be mentioned when appropriate in the Chapters 6 and 7. While the figure may look messy and while the details of the figure are not intended to be readable, shall the individual figure appearances (and the figure numbers) be a memory aid to either recall the context of an earlier discussion or directly jump to these details again. The arrows among figures represent the chain of thought and influence of Part II onto what is discussed in Part III.

Note, there is a complete synthesis of the most important figures of Part II provided as one diagram (Figure C- 1) in Appendix C. We decided to show Figure 4-4 and Figure 4-15 explicitly as part of the *Part III Overview Figure* because they play a key role and because the complete Figure C- 1 would only add irrelevant detail for the purpose of our discussions. Thus, Figure 4-4 and Figure 4-15 can be considered parts of one meta-model (that of Figure C- 1). Figure 4-15 represents an extension of Figure 4-4 but it does not show the still valid relationships of Figure 4-4 again.



Part III Overview Figure: Informal Relationships Among Key Models and Their Concepts of Part II and Part III

6 Quality Attributes in Software Engineering Revisited

Objectives: This Section shows the utility of our philosophical underpinnings and some of the methodological building blocks in their support to explaining currently confused terms and in support to constructing quality models. As the key contribution, this chapter proposes a new approach on how to construct quality models so as to support the identification and classification of quality-carrying properties of systems and components of different nature.

After having read this chapter the reader will:

- know what we suggest as definitions for the existing, confused terms of non-functional property, extra-functional property, etc.;
- know our suggestion of a systematic approach to construct flexible quality models. This approach implicitly copes with the multi-faceted nature of most properties, which stems from their context dependence.

6.1 Proposed Core Terminology in Context with Non-Functional Properties

In Section 3.2 we claimed that various terms are used in software- and systems engineering to denote properties that do not define the primary matter/energy/information processing functions of a system in its operational context – the functional properties. We feel that terms such as non-functional property, extra-functional property, quality attribute, etc. are not used carefully enough. Based on our research, we would suggest the following distinction, if the currently used terms were to stay.

- *Attribute/property* are treated as synonymous and are used in the most general sense as defined by standard dictionaries, e.g.: “a construct whereby objects and individuals can be distinguished” [87], “a quality or trait belonging and especially peculiar to an individual or thing” or “an effect that an object has on another object or on the senses” [84]. In all these standard dictionary definitions, we would replace the term object with component and thing with entity, as per our general definition (Section 4.2.2.1).
- A *required attribute/property* is expressed as a need or desire on an entity by some stakeholder. We may call such a property a *requirement*. Considering the distinction among types of properties (see below), we may have functional requirements, non-functional requirements, etc. Because of the inherent characteristics of a property (recall complexity in 4.2.1.1), a property can be a compound property, and therefore can a requirement refer to a set of required (and possibly contradicting) properties.
- An *exhibited attribute/property* is an attribute/property ascribed to an entity as a result of evaluating the entity. The evaluation may be direct, in the sense that one does some measurement with the entity in question, or it may be indirect. The latter may be the case when we ascribe a property to an entity because we evaluate related artifacts or because someone made us believe that the entity has this (typically conceptual) property, although we can hardly measure it on the entity itself. As an example, consider testing. We cannot directly measure an entity for its test coverage. We must believe some written reports (related artifacts) or trust the words of a tester. An exhibited property may also be ascribed to an entity as a result of some cognitive process (e.g. categorization).

- *Quality*: The totality of exhibited attributes/properties of an entity that bear on its ability to satisfy stated or implied needs, i.e. to satisfy its requirements. Quality thus represents the set of all exhibited attributes/properties that have a relationship to required properties.
- *Quality attribute/property*: Refers to an exhibited attribute/property that is part of the Quality of an entity.
- *Functional property*: Functional properties are used to denote the primary functions of a component/system relative to its operational context and hierarchical level. They reflect the essential behavioral features of a component/system. Functional properties can be observed as responses to stimuli in the component/system's operational context. A functional property refers to a component/system's ability of processing of information (typically in a information system) or processing of matter/energy (typically found in a "hardware" system).
- *Non-functional properties*: Note, we would prefer to abandon this term totally but because it is widely used we would limit its meaning to the following: Non-functional properties relate to all but the functional properties of a component/system. They are the union of extra-functional properties, afunctional properties, and any other conceptual properties that we might ascribe to the component/system because we consider the component/system as part of any conceptual system.
- *Extra-functional properties* (with its synonyms *Ilities* and *Quality of service*): Relate to properties of functional properties. The extra-functional property is "extra" to, or on top of, a certain functional property. Extra-functional properties can be observed together with their functional property as responses to stimuli in the systems operational context. As such, extra-functional properties are predicables of functional properties.
- *Afunctional properties* (with its synonym *developmental* property): The properties of a component/system seen in its development context. That is, the properties typically describe features of the behavior template or intermediate development artifacts. The latter refer to models and representations of what is going to be deployed.

In order to illustrate the use of this terminology with a simple example related to an ATM, consider the following.

Need description: The ATM software must support the cash dispensing of the requested amount of money. Requesting money must be easy and shall not require computer literacy. The ATM software must be testable without an established interaction with the credit card authorization institute.

As-built description: A constructed ATM software supports a user dialog based on a display and a keyboard. The user dialog leads the user through a context-aware dialog to enter the requested amount of money. 100% of the user entries (e.g. amount of requested money) are checked for plausibility. 14 out of 18 user entries are based on a small number of selectable typical options (e.g. 100.- or 200.- or 400.- for the amount request). The ATM software is composed of 12 source code files. The ATM software features a built in test stub for the emulation of the interaction with the credit card authorization institute. It can be activated and accessed through a dedicated maintenance code entered after ATM boot.

The analysis of the statements above with respect to our terminology yields the following insights (see also Table 17): All but one exhibited property are quality attributes in that they relate to stated needs, i.e. to required properties. Hence, all these exhibited properties together describe the quality of the ATM.

Table 17: Example Use of Terminology

<u>Required attributes (requirements)</u>	<u>Exhibited attributes</u>	<u>Quality attribute</u>
<ul style="list-style-type: none"> ▪ Functional: cash dispensing based on cash request 	<ul style="list-style-type: none"> ▪ Cash request dialog 	yes
<ul style="list-style-type: none"> ▪ Extra-functional: easy to use cash request 	<ul style="list-style-type: none"> ▪ Guided cash request dialog 	yes
<ul style="list-style-type: none"> ▪ Afunctional: Testability without interaction with a credit card authorization institute 	<ul style="list-style-type: none"> ▪ Coverage of user entry plausibility check 100% ▪ Options-based user entries: 14 out of 18 ▪ Number of source code files: 12 ▪ Built-in credit card authorization test stub 	yes yes no yes

6.2 A Flexible Quality Model

As a conclusive insight from our discourse into the nature of properties, we claim that fixed quality models, such as the ISO 9126, cannot account for the model variability that would be needed for the big number of possibly different entities (or more specifically components) they should be applicable to. Even in the ISO 9126, software is defined as representing almost any artifact that is related to computer programs, i.e. all the artifacts produced during the development process may qualify as components. Hence, a large number of diverse types of components shall be served by one fixed quality model. Because of this diversity we prefer a flexible but guided quality model construction approach. That is, we prefer a conceptual aid to construct a quality model in a given situation.

Why do we think that a fixed model does not suffice? As we said in Tenet I, the properties and their definitions are invented by humans and serve to explain phenomena of interest. If new phenomena of interest are discovered (or invented), chances are that the existing properties are not sufficient to describe them. We experienced this fact over and over again in software engineering. New concepts, such as classes in OOP, software components in CBSE, or Web services in distributed internet applications introduced new properties that were not relevant before these concepts appeared. For instance, OOP classes brought about a number of new class-based metrics (depth of inheritance tree, etc. [112] 658 ff.). They were not existent (or “invented”) before the class era. Hence, the shortcomings of a fixed model will show latest when it comes to defining measurable quality attributes, as they are called in the ISO 9126. Identification of meaningful, measurable quality attributes (Dromey uses “quality-carrying properties” [39], a term also much to the point, we think) requires defined *contexts* for such attributes.

What constitutes the relevant context of an attribute is an immediate consequence of our basic model of entities, components, and their properties. This was discussed in the context of Figure 4-4. For convenience reasons, we show Figure 4-4 again as part of Figure 6-1 below. If we follow the associations of Property in Figure 4-4, we are lead to the concepts of Observer, Entity (hence to Component), and from Entity to Context (and hence to Context of Component)²⁶. It is these three basic and general concepts that form the context of an attribute and that are at the heart of the dimensions of our Quality Model Construction Framework.

²⁶ One should not mistake the Context of an Entity in Figure 4-4 with what we call the context of an attribute in this discussion.

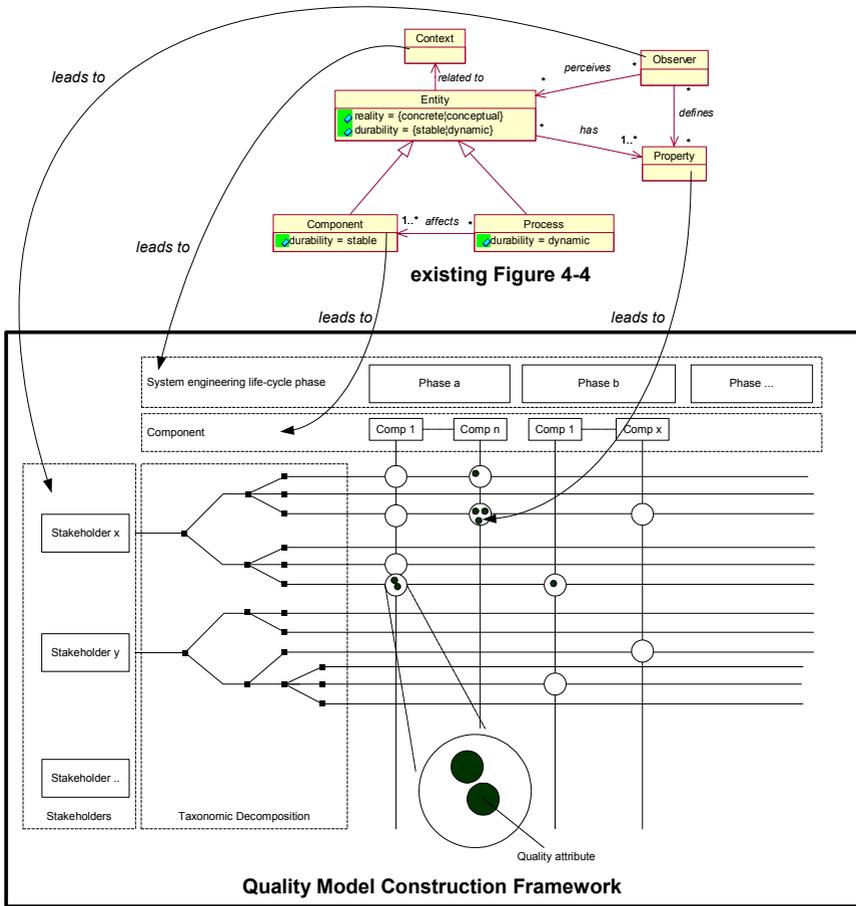


Figure 6-1: Informal Illustration of the Quality Model Construction Framework and its Derivation from the Basic Model of Components and Properties (Figure 4-4)

Figure 6-1 informally shows the “Quality Model Construction Framework” and, by means of arrow-headed lines, how it is motivated by the concepts of Figure 4-4. In the Quality Model Construction Framework we use the term stakeholder, which is derived from Observer in Figure 4-4, we use Component as we did in Figure 4-4, and we use Quality attribute as defined in Section 6.1 and as derived from Property in Figure 4-4. Because of the two inherent characteristics of a property, specificity and complexity (cf. Section 4.2.1), a property may conceptually be composed of properties. This yields the taxonomic decomposition dimension of the framework with its characteristics and subcharacteristics.

The Part III Overview Figure on page 123 shows the same dependence on Figure 4-4. Because Figure 4-4 is the most *general* model that motivates the dimensions of the quality model construction framework, the Part III Overview Figure shows additionally two more key figures of Part II (Figure 4-15 and Figure 5-2). They help to add specificity and practical considerations in deriving the quality model construction framework. The contribution of these figures is made explicit in the subsequent explanations.

In general, we view the flexible quality model as a construction framework for developing customized quality models. A customized quality model is a quality model for a defined set of components with defined quality attributes for defined stakeholders. A concrete example, in which the component is a software component, is given later in Section 7.1.3.

The construction framework consists of the following main dimensions:

- **A set of stakeholders:** Stakeholders are the all important starting point when it comes to defining components and their properties. Stakeholders provide the utilitarian perspective, i.e. they determine how they interpret a property and what they use it for (e.g. prediction or assessment). Because perception is always relative to the observer/stakeholder, what is considered a component and a property *worth perceiving* depends on the observer/stakeholder.
- **A taxonomic decomposition tree of high-level attributes:** The tree represents a hierarchy of quality attribute determinables/determinates called characteristics and subcharacteristics (represented as filled squares in the taxonomic decomposition pane of Figure 6-1). Opposed to fixed models, we allow this decomposition to be specific to stakeholder-type. This acknowledges the fact that the set of characteristics, their decomposition, and their naming is very much dependent on stakeholder community and terminology. A fixed model (such as the ISO 9126) may of course be used as a starting point. Also, it is perfectly conceivable and probably preferable to define a single basic decomposition that might be appropriate for a set of stakeholders. The stakeholder dependency of such decompositions simply says that maintainability, to take one example, is not the same for every stakeholder. Maintainability may have analyzability as an important subcharacteristic for a stakeholder, say a programmer, in the development phase because he may associate with it efficient bug-fixing of source code. But analyzability may not be of interest at all to a stakeholder, say a data typist, who deals with a system in its operational phase. There, maintainability may potentially refer to the ability to easily maintain the customer data in the account management system of a bank. Or, for an IT system administrator, maintainability may even refer to dynamic extensibility of a software application (extensibility is defined in the Appendix Table 25 on page 175).
- **A set of system engineering life-cycle phases:** For partitioning the domain of inquiry, we use the life-cycle phases of the system engineering (or system development) process. Two reasons motivated this choice: (a) Life-cycle phases and components (discussed below) stand for the two main ingredients of a development method: *process, artifact* (cf. Section 2.4); (b) The *domain of inquiry dimension* of Figure 5-2 is a practical choice for *Context* of an Entity as shown in Figure 4-4 (thus the influence of Figure 5-2 on the construction framework as indicated in the Part III Overview Figure). Life-cycle based partitioning helps to identify the components that are relevant in a specific phase and to put them into a defined context, when properties of these components are discussed. Phases can be as coarse-grained as “systems development” and “systems operation”. This is exactly what we used in the domain of inquiry dimension of the 2-2-2 principle in Section 5.1. However, phases may also be finer-grained such as: requirements phase, architecture phase, design phase, implementation phase, testing phase, operation phase, or/and project planning phase, controlling phase, scheduling phase, etc. Note, these

phases may be in different organizations and potentially overlap. For example, operation may be in the business organization that runs a software application, while implementation would typically be in the development organization. However, this distinction does not need a special dimension in our quality model.

- **A set of components:** By *components* we mean the units/elements to which properties can be ascribed. It follows the definition derived from Figure 4-4 in Section 4.2.2.1. As a consequence, we restrict ourselves to the subset of stable entities. This is then in line with the notion of entity as used by the ISO 9126. Following Figure 4-15 (see Part III Overview Figure on page 123), a component in the quality model construction framework can stand for any component with conceptual or with concrete reality. Hence, this includes concrete and conceptual systems and their components and therefore also includes the behavior template and its components. If we constrain our discussion to the ISO 9126 scope, which encompasses *software product* quality, today's relevant components are essentially found by considering the behavior template and its related artifacts and concepts in the life-cycle phases of a software product development. For example, in the requirements analysis phase explicit components/concepts used may be the system to be built, a requirement, a requirement specification (the set of requirements as a conceptual system), a use case, etc. In the implementation phase, a component may be a programming-language specific implementation class, a class method, a parameter of a method, and so forth. Hence, the ISO 9126 entities stand for those components that are treated explicitly in a certain phase and - to become philosophical again - are communicated and exchanged among humans and possibly between humans and computers. But as said earlier, in every life-cycle phase potentially different components are the primary focus of attention. The set of components needs to be agreed upon somehow. The choice of components is normally reflected by and in reciprocal effect with the choice of stakeholders.
- **A set of measurable quality attributes:** The cross-section of a stakeholder-specific leaf characteristic and the component seen in a particular phase defines a set of quality-carrying attributes - the quality attributes per component and stakeholder-specific subcharacteristic. Every such non-empty cross-section is a subset of the set of all quality attributes that are relevant for the particular stakeholder and the specific component as defined in the phase. All the stakeholder/component-specific property sets together determine the final set of measurable quality attributes for the component in question. Sets may and normally will be overlapping. In fact, it is preferred that they are, because it would mean that the cost of measuring a quality attribute is justified because several stakeholders can benefit from knowing its value. Furthermore, if the same type of component is present in more phases, it may potentially carry a greater number, but more importantly, a greater diversity of properties. Because the cognitive interaction of a stakeholder with the component defines the set of properties, interaction scenarios in a very broad sense can be the source to elicit the set of relevant attributes. For instance, while "entering a pin code" is a more obvious scenario, which may lead to some quality attributes (functional and extra-functional ones), "replacing the underlying thermal model of a power transformer" in an online monitoring and diagnostic system of electric power equipment is a more subtle one relating to the maintainability of a diagnostic program. For a systematic approach to derive the measurable quality attributes we suggest to adopt the Goal-Question-Metric approach. The latter is discussed more detailed by means of an example when we apply our construction framework to software components (Section 7.1.3).

Note that this construction model does not say in which sequence the elements of each dimension are to be defined in a given situation. That is, we may use the framework by starting from different corners, e.g.:

- “Given a process life-cycle phase (e.g. requirements analysis), what are the concrete or conceptual components we deal with (and typically share in teams)? If we have identified them, which are the relevant stakeholders that might be interested in these components...”
- Or, “Given a specific component in what life-cycle phases is it present? What are the stakeholders, ...”
- Or, “Considering some high-level quality attributes (e.g. usability), what does it mean for a certain component (e.g. a software component) in the context of a certain stakeholder (e.g. an software architect)...” For instance, usability for an end user could refer to the GUI-related quality attributes of a software component, while it might relate to the integration related quality attributes for a software architect, who associates the ease of reuse with the software component’s usability.
- Etc.

Despite not being prescriptive on how to apply the model, it still defines what dimensions shall be considered and therefore provides the “conceptual fences” (or one could call it a metamodel) for the intellectual task of defining concrete instances in each dimension for a given situation.

It is important to clarify that the phases in our system engineering life-cycle based domains of inquiry are not redundant to the stakeholder dimension. For instance, properties related to a component in the “testing” phase are not relevant for the “tester” stakeholder only. While they certainly are relevant for the tester, there are potentially several other stakeholders who are interested in certain properties related to the testing phase of a component. For example consider the component “test plan”. A property of the test plan is its “being reviewed”. This is not only of interest to the tester but also (and probably even more so) to the quality manager.

Intuitively, of most interest are those components that are relevant to as many stakeholders as possible and that appear in as many phases as possible. This would allow to do research on cross-phase attribute relationships and by this attempt to make correlation statements among properties measured in different phases. Components, which appear in several phases, would also serve as prime candidates or fulcrum points for attaching quality attributes that are meaningful to as many stakeholders as possible. It is this idea, in line with our observation made in Section 4.3.1.3 on the important conceptual and concrete systems in software engineering, which inspired our quality description model for software components discussed in Section 7.

In summary, the variability in our flexible quality model (as opposed to a fixed quality model) comes through the variable set of stakeholders, the variability in taxonomic decompositions, the variability in life-cycle phases considered in the system engineering process, the variable set of components, and finally the resulting variability in the sets of considered quality attributes. The members of all these sets are almost unlimited and hardly any set is precisely defined and agreed upon. This is one explanation why holistic quality attribute discussions are so hard to sustain or why Hochmueller observed that “the structure of extra-functional requirements can soon overcharge our capabilities in dealing with complexity” [55]. However, agreements (such as standards) could be founded based on this structure to limit quality attributes to explicitly defined and limited sets.

6.2.1 Possibility Matrix for Quality Attributes

We can represent the essence of the Quality Model Construction Framework in tabular form to obtain a generic possibility matrix for quality attributes. An example is shown in Table 18. The possibility matrix allows identifying and classifying potential quality attributes and metrics per component and phase under a stakeholder-specific, taxonomic decomposition. The highlighted and with “•” marked cells in a table show for which component in a life-cycle phase there might be (or there are) reasonable quality attributes defined for given stakeholders. Hence, it identifies that at a particular life-cycle phase there is an opportunity for obtaining measurements from a specific type of

component. Among the utility of such a matrix to systematically discover quality attributes, it can also be used to collect and organize existing quality attributes and metrics found in the various literature.

As an example of a possibility matrix consider Table 18, in which only two phases in the development of a software product (say, the ATM software) are shown: Design and Operation. There exist many possible entities (components in our words) to which properties can be assigned. These are for example system, function, use case, class, software component, parameter, etc. For brevity, we chose just two rather diverse ones to make the point of the possibility matrix:

- System: it stands for the software product as the set of deployment artifacts in the Design phase. It stands for the system executing in its operational context in the Operation phase.
- Class Method: in the Design phase, it stands for the OO construct of a programming language class that implements object behavior. It stands for a piece of program that is executed on the target platform in the Operation phase.

We chose two stakeholders (End User and Designer) and only one high-level characteristic (Usability) and two subcharacteristics (Understandability and Operability). The marked cells show that for a System there might be measurements made, i.e. measurable quality attributes defined, in both life-cycle phases. With respect to Understandability both the End User and the Designer might have quality attributes that are of interest to them. However, they must not be the same. For instance, one of the quality attributes related to the Understandability of a System for a Designer might be “the textual annotation of each source file with respect to the system functionality it serves” (marked through cell Design/System:Designer/Usability/Understandability in Table 18). For an End User, on the other hand, “time needed to operate the cash dispensing functionality of the ATM system” might be relevant (marked through cell Operation/System:End User/Usability/Understandability).

For a Designer, Operability may have a different connotation in the Design and the Operation phase. For instance, in the Design phase a Class Method may have a quality attribute that expresses whether the Method can be used (operated) according to an known interface standard. In the Operation phase, operability may refer to an attribute that expresses whether the Class Method supports a way of switching the Method at run-time into a monitoring mode that tracks its operation status.

Table 18: Example Possibility Matrix

		Design			Operation		
		System	...	Class Method	System	...	Class Method
End User	...						
	Usability	...					
		Understandability				•	
	...						
Designer	...						
	Usability	...					
		Understandability	•		•		
	Operability			•		•	
...							
...							

6.3 Standards and Related Research Revisited

Structuring attributes by means of classifications has been pervasive in the quality attribute related research and practice. After the derivation of our flexible quality model we started to understand why classifications were approached from different angles up to now, i.e. why different dimensions could be leading: components, phases, stakeholders, or even classification schemes (such as the ISO 9126). Therefore, our flexible quality model framework can also be used as a birds-eye or a holistic view to position related work and quality models or property classifications that are proposed in research literature or in international standards (as discussed in Chapter 3). In the following, we will therefore revisit some of the work mentioned in Chapter 3 as well as briefly position some important current research work.

6.3.1 ISO 9126 Revisited

The current ISO 9126 defines a fixed taxonomic decomposition for a vaguely defined set of components. To account for this vagueness it must be as generic as possible, which is documented by the rather general definitions of characteristics and subcharacteristics. They are general because the standard does not relate them to specific, defined software artifacts (or components in our words). Defining the missing but promised Parts 2, 3, and 4 will therefore be a challenging task. These Parts will contain quality attribute metrics to quantitatively measure *external*, *internal*, and *quality-in-use* software product quality, respectively. Unless the Parts will not define a fixed set of components that together represent a software product, it will be hard to provide an exhaustive set of measurable quality attributes with their metrics. On the other hand, if the standard defines a fixed, but for practical reasons a small set of components with their proposed quality attributes, chances are that it is outdated before it is effectively used. This would mean that potentially new types of components are not covered. Hence, what would be needed is a starting set of components and quality attributes and a standardized way on how to extend the set of components and quality attributes in a standards-conformant way. A meta-model would therefore be of great help. The model provided by the Part 1 (Figure 3-5) is not powerful enough.

Our flexible quality model represents an extension to the current model in the ISO9126 Part 1. It would rigorously define the structure of the remaining Parts 2, 3, and 4 and provide for their guided extension or customization. We can extend the ISO9126 quality model of Part I (the essence of which is drawn in Figure 6-2 on the left) to the meta-model shown on the right side of Figure 6-2. This amounts to taking our flexible quality model framework and constraining it with the ISO 9126 normative aspects. The constraint mainly refers to the closed set of characteristics and subcharacteristics.

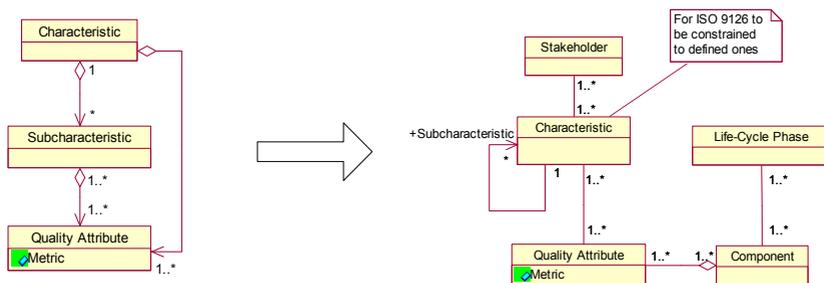


Figure 6-2: Extending the ISO 9126 Quality Model

The very recently available recommendations for Part 2 and 3 [64, 65] confirm our proposal. The external and internal metrics, which are defined in these two reports have introduced the notion of “life-cycle phases” and “target audience” in order to provide contextual information for the explanation of the metrics. Hence, our model concepts of *life-cycle phase* and *stakeholder* seem to be important indeed. Unfortunately, the ISO 9126 does not make these concepts visible/explicit in its quality meta-model. Further, we can say that the draft Parts 2 and 3 do not explicitly relate the metrics to the types of components they belong to. The type of component can only be inferred through interpretation of the purpose of the metric.

In summary, our proposal is backward compatible with, i.e. subsumes the meta-model of Part 1 and can be used and would even be compatible with the early drafts of the standard’s Part 2 and 3. Part 2 and 3 explicitly mention: “*These Technical Reports...provide a suggested set of software quality metrics (external, internal and quality in use metrics) to be used with the ISO/IEC 9126-1 Quality model. The user of these Technical Reports may modify the metrics defined, and/or may also use metrics not listed. When using a modified or a new metric not identified in these Technical Reports, the user should specify how the metrics relate to the ISO/IEC 9126-1 quality model...*”. Hence, if a model like our suggestion were available, new components and quality attribute metrics could be orderly added and related to the quality model.

A Note on the other standards: IEEE Std. 1233, ANSI/EIA 632, IEEE Std. 830

All three standards define a flat categorization of (required) properties. With respect to our flexible quality model, they define a fixed taxonomic decomposition over one level only, i.e. a list of characteristics. Further, none of these standards has an underlying (explicit or implicit) quality/property meta-model.

6.3.2 Related Work

Based on our flexible quality model we may also position related work:

- The research work towards a software component quality model, which was/is carried out by Bertoa et al. [17] [79], propose four dimensions: stakeholder, quality characteristics, life-cycle phase, and granularity/visibility. The latter dimension amounts to a system vs. component (or inside vs. outside) perspective. Their model is close to the ideas of our flexible quality model. We may say that it was partly influenced by our ideas. We deduce this because of their references to our work, in which they state that they derived a classification dimension from our ideas.
- A reduced form of a possibility matrix was used by Purao et al. [113] in the context of OO metrics classification. They defined two dimensions (phase, and OO-entity) as opposed to our four major framework dimensions (phase, component, stakeholder, and taxonomic decomposition). Their recent and comprehensive study made the observation and proposal for the importance of seeing OO artifacts in the context of life-cycle based domains of inquiry (or “states” as they call it). They used this observation as the basis to identify which OO artifacts (class, method, link, package, etc.) are relevant in what states and thus could carry life-cycle specific quality attributes.
- A lot of existing work has been carried out to define the relevant quality attributes and most importantly their metrics for a *restricted number* of component types at a *particular* phase. For example, [11] or [39] use programming language components to assess program design quality; or [77] defines properties to assess the component of a requirement, which is a conceptual component of a requirements specification. Most of such component-exclusive work mentions the constrained (one type of component in one life-cycle phase) applicability of the proposed metrics. By this, they miss out on the chances that possibly different other measurements could have been made, if the component had been mapped against other phases.

7 Quality Description of Software Components

Objectives: This Chapter presents how the quality of software components can be described in a structured way considering the research work presented thus far. In particular, the kinds of quality attributes to be described are a direct consequence of the three types of important systems and their related quality domains (Section 4.3.1.3). Further, the identification and structuring of quality attributes makes use of the flexible quality model introduced in Section 6.2. The application of the conceptual structure shall be documented by its integration into the Reusable Asset Specification (RAS).

After having read this chapter the reader will:

- know how we structure quality attribute descriptions for software components;
- know how we extend the logical model of the Reusable Asset Specification (RAS) with such descriptions;
- be aware of the XML schema alternatives we suggest for a RAS physical model and which one we chose for describing specific ABB profiles;
- be informed about the usage of the RAS in the context of ABB reusable assets.

7.1 Data Sheet for Software Components

In 1999 we argued for four areas of potential promises of CBSE and its use of software components [7]: software components to raise the programming abstraction, software component technology to improve application interoperability, software components to support a (visually driven) compositional programming style, and software components to facilitate software reuse. While the former three promises have materialized to a large extent, has a component-market based approach to reuse only had limited success. In a recent survey [51] (which also compares its results to previous surveys), the trust in the quality of a software component was voted the second greatest barrier of component-based software engineering. The highest barrier is lack of training in using components. In this Section we focus on this issue trust into component quality and suppose that for reuse to happen, there must be *well-described* reusable assets available. A software asset can be informally defined as a partial solution packaged for reuse to expedite the development of software that incorporates such an asset. In our case, these assets are software components or sets of software components. The focus in this Section lies on the description of non-functional properties of software components. By having such a description we expect to improve the level of information about a software component and thereby eventually reduce one barrier to component-based reuse.

The structure of the proposed description is a concrete application of the quality model construction framework discussed in Section 6.2 (see also Part III Overview Figure) to create a software component specific quality model. Its concreteness stems from the focus on one single type of component – a software component – and on the pragmatic choice of three life-cycle phases. The pragmatics behind the latter choice is motivated through the three important types of systems, which

we identified in Section 4.3.1.3. This is indicated through the impact of Figure 4-11 in Part III Overview Figure.

In general, systematic software reuse involves at least two distinct roles: software asset producers and software asset consumers²⁷. This is exactly the main characteristic of a CBSE development process too. While a development process in CBSE is in principle no different to other software development processes there is at least one difference to be stressed: the development *of* components and the development *with* components. In an envisioned component market, these two processes tend to be physically and timely decoupled and carried out by different organizational units or companies. Consequently, we can partition the development life-cycle phase at least into two main blocks: *producing components* and *consuming or using components*. In order for a consumer to reuse software assets such as a component, she must be able to identify potential reusable assets, evaluate their suitability, obtain all the relevant artifacts that are needed to properly (re-) use a software asset, and have a well-described instruction on how to reuse the partial solution. These consumer-oriented concerns and how an asset description can principally cope with them are shown in Table 20.

Table 19: Asset Consumer Concerns and Solution Approaches

<i>Consumer Concern</i>	<i>Solution Approach</i>
How to discover assets	Classification mechanism
How to select specific assets considering asset features, quality, and constraints	Content description
How to obtain all artifacts of an asset and apply (reuse) an asset	Usage description
How to read and interpret the selected asset description	Precise and unambiguous description model

The evaluation of the suitability is preferably based on an explicit “data sheet”, i.e. a reliable description of the quality attributes of the asset. Such a data sheet shall serve as the basis for the contract between the producer and the consumer. Of course, not only the evaluation of the suitability but all four aforementioned activities requires information that is prepared for the consumer in a coherent and structured way. The Reusable Asset Specification, discuss later in Section 7.2 below, is an attempt to do exactly this: specify a structure to describe reusable assets.

7.1.1 Software Component Contracts

The reusable asset types of further interest to us are software components. Hence, we focus on component-based software reuse and thus on data sheets as the name for quality attribute descriptions of software components. In fact, the data sheet represents the embodiment of a component “interface”, where interface is used as the generic term for a specification of assumptions that a consumer may make about the component advertising such an interface [102]. The assumptions in turn rest upon the described properties ([136] p.55). Note that this definition of interface is richer than the narrow application programming interface (API) meaning as for instance defined in the UML and used in everyday component programming.

This generic definition of interface comes closer to what others termed component contracts [18, 114] or simply component specifications. The notion of *contract* for programming artifacts has been formally introduced by Meyer as part of Eiffel [85]. The basic idea, taken from Hoare’s axiomatic

²⁷ In the context of a more elaborate, systematic reuse process, more roles are typically defined to hint at the specific activities they have to perform in a reuse process [67].

basis of programming [54], was that of specifying object behavior with assertions such as pre- and post-conditions. Since then, other types of contracts have been proposed. It is not only the kinds of properties that determine the types of contracts but also the implicit assumptions of who are the stakeholders involved in the contract. It is therefore important to mention that the notion of contract, as introduced by Meyer and later extended to software components by Beugnard and colleagues [18], uses the term *client* in the restricted sense of representing the piece of software that will programmatically use the component (also seen as a piece of software). Hence, the client is *not* the consumer of a software component in the reuse sense of the term, although a component integrator (as one important *consumer*) is of course interested in many client-programming related properties. Only the 7-level model (discussed below) implicitly weakens the definition of a client in that the contract extensions clearly address properties that are relevant for humans as clients, i.e. they address more than the programming-related concerns of the consumers. Although a stakeholder centric definition of properties would clearly avoid confusion, neither of the contract models makes stakeholder-relations explicit.

The 4-level contract. In [18] [114], four types (or levels) of contracts for software components are identified: basic or *syntactic*, *behavioral*, *synchronization*, and *quality-of-service*. Today, the specification and usage of syntactic contracts relies on the likes such as Interface Definition Language (IDL) and C++ header files. Behavioral contracts are an integral part of some languages (like Eiffel [85] or contract extensions to Java [144]) and rely on pre- and post condition, and invariant formalisms. In essence, they specify what information processing behavior the client can expect on proper invocation of a single method. Synchronization contracts refer to the definitions that shall specify how a software component can cope with parallelism and multiple clients. Synchronization is not a new problem and has been dealt with in various domains in computer science. The approaches proposed for software components follow the idea of a process algebra [25] [138]. In [110], we suggest to extend synchronization contracts to not only specify parallelism issues, but also explicitly specify semantically coherent component usage scenarios, i.e. to define the sets of permissible sequences of operation invocations. Finally, quality-of-service contracts specify what we called extra-functional properties. As can be inferred from the description of these four levels, their scope of properties is very programming-centric. That is, they address the specific assumptions that typically a programmer may want to make of the component in order to explore its behavior in an interaction of components or even stronger, they address assumptions that a runtime verification mechanism could verify.

The 7-level contract. The 4-level approach has recently been generalized and extended by a memorandum for the specification of business components, filed by the GI (Gesellschaft für Informatik) [46]. The component specification defined by the GI now includes seven levels (Figure 7-1). Compared with the 4-level model the synchronization level has been generalized to *interaction level* and now also covers the specification of patterns of interactions (very similar to [110]), and the quality-of-service level has been generalized to *quality level* and includes non-functional properties that go beyond QoS in that now also afunctional properties are included. Further, the top three levels are new and accommodate “other” properties deemed important. We say “others”, because their existence and contents seems somewhat at will. It would have been much more intuitive if a stakeholder-related definition had been given. Nevertheless, they clearly indicate that more than programming relevant properties need to be addressed. All the properties found in the last three levels are conceptual properties as per our definition. The *terminology level* contains essentially a dictionary of terms used in the other levels. The *task level* specifies in which business processes and tasks the component might be of use (e.g. “useful for money transaction processes”). It therefore serves a classification purpose. The *market level* finally specifies properties that are of primary concern in a component marketplace (e.g., who is the vendor, price, etc.).

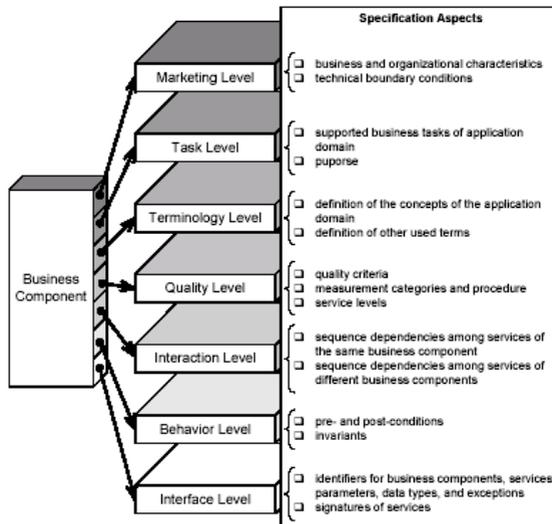


Figure 7-1: Specification Levels and Aspects of Business Components (Source [46])

We will not go into the details of how properties on each level are represented. This can be found in [46]. However, as far as the quality level is concerned neither the 7-level model nor the 4-level model define a quality attribute model or a quality attribute representation.

Our quality description, proposed in the next Section, fits to the quality level and suggests a structured approach to represent its contents.

7.1.2 Conceptual Model of Quality Description

The definition for a system, which is presented in the beginning of Section 4.3, calls for the identification of the relevant units that are part of a system. It is interesting to notice that the concept of a software component is a prominent unit in all of the three important systems that were mentioned in Section 4.3.1.3 (development group, software system as a behavior template, and computer system). We further said that these three systems directly relate to the three quality realms in software engineering. This fact makes a software component a powerful concept not only to alleviate model and view mapping problems, but also to serve as a fulcrum point in linking together these quality worlds. More specifically, a software component is:

- a) the unit of development, of integration, of reuse, and of versioning in the development project or group. This covers both, the development *of* components and the development *with* components. Hence, it may carry development process related qualities.
- b) the unit of deployment and the unit of which the program source code is composed²⁸: it is a unit of the behavior template. Hence, it may carry behavior template properties, or in software terminology: static software product qualities.
- c) the unit of instantiation and execution and of service provision on the computer system in the deployment environment. Hence, it may carry execution-related qualities.

²⁸ It is also a prominent unit in many design models (i.e. conceptual systems) above the program code.

This reasoning led us to our hypothesis that a software component is the one concept that is explicitly present in, i.e. it is a component of, several phases in our dimension “system engineering life-cycle phases” of the flexible quality model. A software component thus has the ability to carry quality related information that is relevant for different purposes and stakeholders. It can support those that implicitly expect software artifact quality from development process quality, those that expect development economics from the static qualities of software artifacts, and those that are interested in reasoning about runtime assembly-level properties based on software component properties [35]. Based on this, we propose a model for the logical structure of a quality description of a software component as depicted in Figure 7-2. This model is a justified instance of our flexible quality model depicted in Figure 6-1 (see also Part III Overview Figure):

- The type of component to carry quality attributes is the software component
- The life-cycle based phases are development process of the component (captured under Process Quality in Figure 7-2), design/implementation of component (Static Product Quality), and execution in deployment environment (Quality of Service).
- Unless otherwise stated, the stakeholders are by default technically oriented component consumers, i.e. system integrators or also called system architects.
- The taxonomic decomposition of the flexible quality model is realized through the nested Quality Characteristics in Figure 7-2.
- The quality attributes are defined through their <name, value> pair, which is related to a Metric and optionally a Scope.

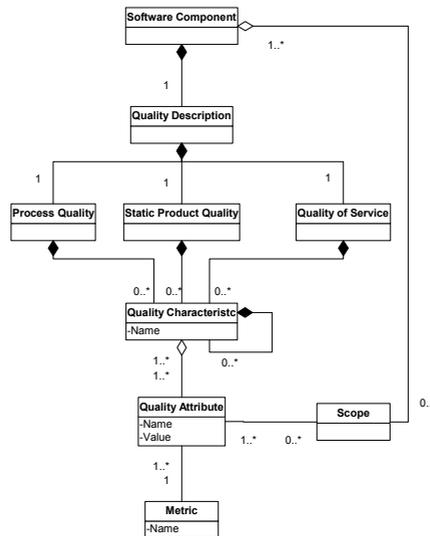


Figure 7-2: A UML Metamodel for the Quality Description of a Software Component

The content of Figure 7-2 can be paraphrased as follows:

- Conceptually, a software component has one Quality Description that is composed of three sections: *Process Quality*, *Static Product Quality*, and *Quality of Service*.

- Each section consists of an enumeration of *Quality Characteristics*, each of which relates to a set of *Quality Attributes*.
- Every quality attribute relates to one particular *Metric* and it can be associated with one *Value* on some scale out of the range of possible values defined in the metric. A value may of course be a complex structure and accommodate for things such as confidence intervals, tolerance, and others. Metric is defined in the ISO 9126 as “the measurement method and measurement scale” ([63], p. 20).
- By default, every quality attribute is defined in the context of the Software Component (implicitly derived through the aggregation hierarchy) and in the context of one or more (sub)-characteristics of one quality section. A quality attribute can be related to optional further Scope(s). Scope is the generic concept to capture the fact that a particular quality attribute may provide contextual information that goes beyond the default. Intended usages of Scope are:
 - A quality attribute may be associated to a software component’s syntactical elements (interface, operation, parameter), in which case these elements represent the relevant Scope. This is close to the idea presented by Lüders et al. [75], where credentials (which is a <name, value, credibility> triple) can be associated to component, interface, and operation. For us, also parameter is an important concept, because quality attributes such as “encrypted data representation” or “compression” might be associated to individual parameters. As a concrete example for such a scope consider a quality attribute (say latency). It is typically specified for a certain operation only. It is thus defined in the scope of one specific interface and one of its operations. One will notice that these syntactical elements are no more than conceptual components of a software component seen as a syntactical system. Thus we may actually ascribe quality attributes to components of a software component.
 - A quality attribute may be related to a certain usage pattern it is dependent on. In this case the usage pattern provides further context information. Such a pattern could represent a permissible method invocation sequence, or it could reference a collaboration in which the software component is a collaborator of. Therefore, an attribute value may even be a textually described response to some stimulus, where the stimulus might be described in the context.
 - A quality attribute may be related to a rely/guarantee type of construct. For example, the value for the response time attribute is *guaranteed* to be “better than x milliseconds” if the software component can *rely* on not being preempted while executing the operation.

Since the model in Figure 7-2 is only a metamodel using general concepts, the latter must be specialized to be instantiable for a software component at hand. That is, the definition of concrete characteristics, attributes, and metrics for the case at hand is to be made. For this purpose we make use of our quality model construction framework.

7.1.3 Determining Quality Attributes

With regard to the customization of a quality model based on the quality model construction framework (as discussed in Section 6.2), we have already chosen the component, the life-cycle phases in which the component is to be seen, and the stakeholders. If we now determine the relevant quality characteristics and possible taxonomic decompositions, we have scoped the possibility matrix of quality attributes. For the definition of a taxonomic decomposition, we use the ISO 9126 model of characteristics as a starting point. In the context of a stakeholder-dependent subcharacteristic and a life-cycle phase we can then start asking what kinds of assumptions the stakeholder might want to make of the component, i.e. what quality attributes would he expect to be advertised. In order to ask

these questions we may draw upon existing experience, interview stakeholders, or/and follow a systematic approach similar to the Goal-Question-Metric (GQM) paradigm. We will show a compilation of quality attributes for software components later in this Section in Table 21.

The GQM was originally suggested by Basili and colleagues [13] and is applied still today [74]. For us, the GQM method translates into: “what measurable quality attributes (M) help answering what questions (Q) a stakeholder wants to have answered to be able to determine whether her goals (G) are or will be met”. We are interested in specifying the quality attributes to be included in a data sheet that help the evaluation of the component suitability from a non-functional property perspective. For this reason, we focus on the *system architect* and *system project manager* stakeholders. They represent evaluators which are on the consumer side of reuse. In order to not complicate this discussion by introducing further stakeholders, we assume that the project manager has full commercial responsibility for the project and the final system. As a simple and almost self-explanatory example of our GQM utilization see Table 20.

Table 20: Deriving Quality Attributes from Stakeholder Goals Using the GQM Paradigm

System Architect	System Project Manager
<p>Context: (Phase) Execution (Characteristic) Efficiency->Time behavior</p>	<p>Context: (Phase) Development process overall (Characteristic) Process quality-> Standards conformance</p>
<p>Goal: (Gyyy-1) Predict the picture update rate of dynamic items on a process control screen.</p> <p>Questions: (Qyyy-1) What is the throughput of the two candidate software components for the protocol stack of our operator workstation?</p> <p>Measurable quality attributes: (Myyy-1/Qyyy-1) Externally procured software components must specify throughput under defined conditions. => tangible property to be defined in data sheet: <i>(Message throughput, msg/s, context¹⁾)</i></p> <p>¹⁾ context shall specify for which type of message this is valid (e.g. continuously polled process data) and/or for which interface/method. Context shall also specify what the environmental assumptions are to guarantee the specified value. Alternatively, it shall state how the value was obtained (which would then cover most of the aforementioned possibilities for context)</p>	<p>Goal: (Gxxx-1) Deliver a system that is compliant with the regulations for systems delivered by ISO9000 certified companies (as specified by the customer).</p> <p>Questions: (Qxxx-1) Are all suppliers of the externally sourced components ISO9000 certified? (Qxxx-2) Do we have a valid ISO9000 certification?</p> <p>Measurable quality attributes: (Mxxx-1/Qxxx-1) Externally procured software components must specify (convincingly) whether producing company is ISO9000 certified. => tangible property to be defined in data sheet: <i>(ISO9000 certified company, Yes/no, proof²⁾)</i></p> <p>²⁾ proof may specify the audit reference or even refer to a copy of the certificate.</p>

For each of our two stakeholders, we list one possible goal from which, over questions, quality attributes are derived that are worthy to be published in a software component data sheet. Note, typically a property-value pair will not do. Often, there need to be additional information available (in

the example in Table 20 called *context* or *proof*) to deal with the fact that hardly any property is contingent and/or that we can not simply assert truth to the property-value pair without some form of supporting evidence to gain trust in the stated value. This is why Shaw [125] offered the idea of a *credential* as a list of properties, each property having the form of a triple (property, value, credibility). Credibility is therefore her term of choice for what we called context or proof in the example in Table 20 and for which we intended to use the Scope concept in Figure 7-2.

In Table 21 we propose a number of meaningful (and not too exotic) non-functional, quality attributes that a potential consumer would wish to find published as part of a software component description. We compiled the list based on our own experience in component-based development (“wish-list”) and adopted or modified existing, but not necessarily software component-related, literature [17] [45] [112] [113]. The table is structured according to our metamodel of Figure 7-2. It contains extra-functional and afunctional as well as some conceptual properties.

Evaluating the suitability of candidate components is assumed to take place on the basis of the description of functional and non-functional properties. The suitability measures themselves, such as proposed in [136] [17], are therefore derived measures (or properties) that represent some form of relationship between published quality attributes and required properties. For example, the suitability property *coverage* is defined as the percentage of matching provided interfaces of a candidate software component versus the total number of needed interfaces [17]. As such, coverage like any other suitability property cannot be part of a component data sheet, although they are part of the ISO 9126. Finally, it is interesting to mention that software component descriptions, which are currently available on software component marketplaces, are far from explicitly providing the kinds information we would like to assume about software components, let alone in a structured way as discussed in this Section [16]. Further, our own experience in designing a schema with properties for reusable software components within a global company shows that (a) even the agreement on just a few elementary quality attributes is a very demanding undertaking, (b) the types of selected attributes are very much domain dependent, and (c) implementing a process in which component developers actually provide this information, if not generated automatically, requires endurance and full commitment by many stakeholders and decision makers. To secure the latter, sound business cases are a must, but cannot always easily and crisply be made for software component reuse.

The following remarks with respect to Table 21 apply:

- The Scope column refers to the syntactic elements of a software component as mentioned above: component, interface, operation, parameter. They are shown later by the aggregation hierarchy of Figure 7-5 on page 150. Whenever there is more than one element mentioned, the attribute may refer to either of these elements. Although it might typically be most appropriate, and would be preferred, for the element with the finest granularity (e.g. “operation” when the Scope cell says “component, interface, operation”), one would already obtain helpful information if the attribute were ascribed to an element at lower granularity level (e.g. interface). As an example consider the quality attribute *functionality-security:authentication* in Table 21. It would be nice to know that authentication is required for using certain specific operations. However, knowing that authentication is supported for an interface or even for the component helps already to narrow down the potential candidate components.
- Some attributes (like number of provided interfaces) can be obtained if the software component is available and carefully examined. However, because of the expressiveness of such an attribute and the frequently painful work to obtain that value, we consider such attributes worth to be made explicit. Just imagine one had to find the number of supported interfaces of a Microsoft COM component packaged as a DLL (Dynamic Link Library) with a DLL-included type library. Without extra documentation or installing the component on a machine and then examining it in the Windows operating system registry, this can hardly be done.

- In order to not overload the table we have omitted further Scope options, such as credibility or proof, as was discussed in the context of the GQM example in Table 20.
- In the Process Quality section we have used the term reusability (which is missing in the ISO 9126) as a substitute for usability. We have done so, because in the context of components, reusing a component can be viewed as *using* it. Further, usability in the ISO 9126 refers to the end user using a system and as such refers the user interface attributes mainly. In the context of software component reuse, we do not pay special attention to this connotation of usability.

Table 21: Suggested Quality Attributes to be Published in a Software Component Data Sheet

Characteristic – Subcharacteristic	Quality Attribute	Metric	Scope
<i>Quality of Service Attributes important for stakeholder: System architect</i>			
functionality – accuracy	resolution	the degree of discrimination with which a quantity is stated. Measured as percentage to a reference value	parameter (out)
functionality – accuracy	precision	quantitative measure for the magnitude of relative error. Measured as percentage to a reference value.	parameter (out)
functionality - security	auditability	indicates whether component provides logging capabilities of component usage. Value should be Boolean with text string indicating the type and coverage of logging.	component, interface, operation
functionality - security	encryption	indicates whether encrypted data can be handled. Value should be Boolean with text string indicating the type of encryption	parameter
functionality - security	authentication	indicates whether authentication is supported. Value should be Boolean with text string indicating the type and coverage of authentication.	component, interface, operation
functionality - security	denial of service	indicates whether the component protects its services to withstand denial of service attacks. Value should be Boolean with text string indicating the type of denial of service attack prevention	component, interface, operation
reliability - maturity	parameter checking	indicates whether input parameters are checked for reasonable values (e.g. defined bounds) at runtime. Value is Boolean.	parameter
reliability - maturity	exception handling	indicates whether and what kinds of exceptions are handled during the execution of the software (e.g. invalid parameters, unsuccessful calls to dependent components, insufficient memory availability, etc.). Value is Boolean with text string enumerating the kinds	component, interface, operation
reliability - maturity	exception reporting	indicates whether and what kinds of exception reporting is supported. Value is Boolean with text string enumerating the kinds	component, interface, operation
reliability - recoverability	transactional	indicates whether a component provides transaction support for its operations. Value is Boolean	component, interface, operation
reliability -	persistence	indicates whether a component provides state persistence.	component, interface,

recoverability		Value is Boolean.	operation
reliability - testability	supervision	indicates whether and which forms of self-supervision the component has (e.g. start-up verifications of dependent interface availability, watch-dog functionality, etc.). Value is Boolean with text string enumerating the kinds	component, interface, operation
usability - administrability	effort	the initial and continuous effort needed to administrate the component in its deployment environment (e.g. is a COM component self-registering). Measured as two time intervals	component
efficiency - time behavior	responsiveness	time duration from receiving a stimuli until a valid response is produced. Measured as average time with statistical deviation	operation
efficiency - time behavior	throughput	the quantity of information that can be processed in a given period of time, e.g. measured in output values produced per operation with statistical deviation	component, interface, operation
efficiency - time behavior	capacity	the quantity of stimuli a component can process in a given period of time, e.g. measured in operation activation/sec or event/sec	component, interface, operation
efficiency - resource utilization	dynamic memory	defines the amount of dynamic memory (e.g. heap space) needed	component, interface, operation
efficiency - resource utilization	static memory	defines the amount of static (e.g. disk, ROM) memory needed by the component	component
efficiency - scalability	number of clients	quantifies the number of clients a component instance can serve	component, interface

Static Product Quality Attributes important for stakeholder: System architect

functionality - interoperability	data compatibility	indicates whether and which data type standards the component is compatible with. This can be on the level of VB data types supported for COM interfaces, or ASN.1, etc. Value is Boolean with text string enumerating the kinds	parameter
functionality - interoperability	component model compatibility	indicates whether and which standard component model the software component is compatible with (this can be on the general level of COM or domain model level such as OPC). Value is Boolean with text string enumerating the kinds	component
functionality - interoperability	interface compatibility	indicates whether and for which technologies IDLs are available. Value is Boolean with text string enumerating the kinds	component
reliability - maturity	defect density	quantitative measure for number of known defects relative to product size (e.g. known bugs/LOC)	component, interface, operation
reliability - compliance	reliability compliance	indication of whether component adheres to reliability standards. Value is Boolean with text string enumerating the kinds	component

efficiency - size	source code size	Lines of source code (LOC)	component
reusability - dependency	interface dependency	number of required interfaces the component (or interface or operation) depends on	component, interface, operation
reusability - understandability	API documentation	Indicates whether and to what extent IDL-based textual documentation coverage of API operations is available. Measured as triple (Boolean, percentage of operations documented, average length of documentation)	component
reusability - learnability	time to reuse	average time estimated for a reuser to properly reuse the component	component
reusability - changeability	variability	total number of component binding-time specific variation points (vp). Measured as triple (# of design time vp, # of deployment time vp, # of runtime vp)	component
reusability - changeability	variability ratio	the relative number of operations that can be varied in their behavior. Measured in percentage of operations	component, interface
reusability - complexity	number of provided interfaces	quantitative measure which specifies the number of provided interfaces	component
reusability - complexity	number of required interfaces	quantitative measure which specifies the number of required (dependent) interfaces	component, interface, operation
reusability - complexity	operation ratio	quantitative measure which specifies the average number of operations per interface (sum of operations divided by sum of interfaces)	component
reusability - complexity	parameter ratio	quantitative measure which specifies the average number of parameters per operation (sum of parameters divided by sum of operations)	component, interface

Process Quality Attributes important for stakeholder: System project manager

development effort	manpower	development effort to develop (design, implement, test and fully document) the component from scratch. Measured in person-days	component
development effort	skill level	subjective measure that indicates on a scale level how skilled the developers must be to develop such a component from scratch. Measured as a tuple (programming skills, application domain skills)	component
process maturity	certification	indicates whether and according to which development process maturity scheme the development organization is certified. Value is Boolean with text string listing containing the scheme(s)	component
process maturity - testability	coverage	indicates whether and to what extent the current version has been fully tested (regression tested). Values is Boolean with test coverage of functionality in %	component

reusability - understandability	documentation	indication of whether and what kind of supporting documentation is available (e.g. Tutorial, programmer's guide, UML models, sample application). Value is Boolean with text string enumerating the kinds	component
functionality - interoperability	backwards compatibility	indicates whether and with which previous versions this component is compatible. Value is Boolean with text string enumerating the kinds	component, interface, operation
functionality - compliance	external certification	indicates whether and by whom the software component is externally certified. Value is Boolean with text string enumerating the certification authorities	component
reliability - maturity	defect removal	quantitative measure for the average number of removed "bugs" relative to all known bugs between two versions	component, interface, operation
maintainability - evolvability	marketed versions	quantitative measure for number of versions that have been distributed on the market	component
maintainability - evolvability	volatility	quantitative measure for average time between versions	component, interface, operation
maintainability - stability	interface stability	quantitative measure for the average number of interfaces that have <i>changed</i> between two releases	component
maintainability - stability	operation stability	quantitative measure for the average number of operations that have <i>changed</i> between two releases.	component, interface
maintainability - stability	parameter stability	quantitative measure for the average number of parameters that have <i>changed</i> between two releases.	component, interface, operation
maintainability - testability	test harness	indication of whether and what kind of test facilities are or would be provided as part of the component	component

7.2 Extending the RAS Metamodel with Quality Descriptions

In this Section we present how we extend the Reusable Asset Specification (RAS [116]) with quality attribute descriptors for software components as defined in the previous Section. In order to do so, we will first introduce the RAS and its conceptual (or so-called *logical*) model in Section 7.2.1. We then show how we conceptually extend it with quality attribute descriptors (Section 7.2.2). For the physical model (i.e. an electronic format to package reusable assets and their description) we suggest to use an XML-based approach. While the RAS is not normative with respect to a physical model, it hints at the XML, but leaves the details on how to use the XML open. For this reason, we have compared different possible XML approaches in Appendix B.

As in any business setting and as already mentioned earlier, software-related reuse requires at least two roles to be present: an asset producer/provider and an asset consumer. Since the particular pair of business partners is dynamically changing and even discovered during the software production process, one of the fundamental prerequisites is an industry agreement on a common way of describing reusable software assets. The Reusable Asset Specification (RAS [116]) is an attempt to reaching such an agreement in that it is a specification for the structure of reusable asset descriptions. It primarily supports the asset consumers and their concerns, which we expressed in Table 19 of

Section 7.1 above. The RAS is proposed by a consortium of mainly CASE tool developers to be able to exchange development artifacts.

The primary contribution of the RAS is the definition of a conceptual model for the description of reusable software assets. Asset and asset package is used synonymously in the RAS and is defined as "...a set of software artifacts that have been created or harvested with an explicit purpose of applying it repeatedly in a subsequent development effort..." [116]. Assets simply refer to partial solutions of value that are packaged for reuse. They can be of different granularity, support different phases in a development life-cycle, and support different degrees of variability for customization purpose. Some possible types of reusable assets are:

- Design patterns; generic solution patterns to reoccurring design problems
- Executable code modules, for example in the form of software components, or code libraries (e.g. a math library for matrix calculations)
- Frameworks; a large body of design work, sometimes in the form of skeletal systems with a set of services and defined variability points for application customization.

The RAS logical model is represented in the UML class notation and called the *RAS core metamodel* (represented as a UML Package in Figure 7-3). Besides the RAS core metamodel, the RAS also suggests the definition of *concrete* models. They are derived from the RAS core metamodel and introduce additional semantics to describe specific types of reusable software assets such as the ones listed above. These concrete models are called *RAS profiles* and constitute an extension/specialization of the RAS core metamodel. As indicated in the conceptual layer of Figure 7-3, an individual asset is then described by an instance of a profile. As an example, consider spell-checkers. A RAS profile for spell-checkers could be defined according to the rules of the RAS core metamodel. Any company that provides spell-checkers could document their specific marketable version of a spell-checker based on this defined profile and thus could make their product available and easily comparable on any of the spell-checker market places.

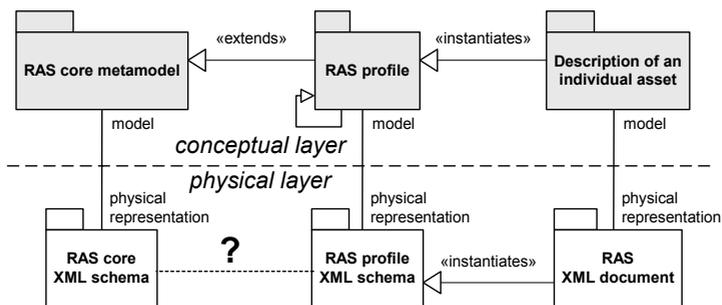


Figure 7-3: Conceptual and Physical Layers of the RAS

The RAS logical model describes how the elements of an asset package are logically structured. But in order to electronically transfer asset packages and electronically process their contents, we should also know what physical pieces make out these logical elements and where this pieces are located. The model of the physical asset structure is called the *RAS physical model*. In a very pragmatic terminology, we would say that we need to know what files (and file sections) contain what logical elements and where are these files and parts located in the universe of accessible machines. The physical model, and the physical layer of Figure 7-3, is discussed later in Section 7.2.3.

7.2.1 The RAS Logical Model

The RAS logical model defines a logical asset structure that must have four parts or sections (see Figure 7-4):

1. *Overview*: The overview part consists of a set of informal, human-readable *artifacts* that describe the software asset and the problem it solves. The artifacts can be in the form of any document (e.g. a brochure), references to web pages, etc.
2. *Classification*: The classification part contains a number of descriptors, basically name/value pairs, to facilitate the classification of software assets. The main objective is to simplify search, browsing, and retrieval of assets. The descriptors in a classification would typically classify the entire asset and not individual components (or artifacts to use the RAS terminology) of an asset. However, the RAS allows classification extensions, which are made in profiles, to refer to artifacts of the solution section. This is our means to include quality attributes, as we shall see later.
3. *Solution*: The solution part is composed of the set of artifacts that constitute the core of what a consumer was looking for - the actual things that shall be reused. These can be artifacts such as design models, source code, executable software components, test procedures, installation scripts, etc.
4. *Usage*: The usage part contains the information that is needed to properly apply the solution artifacts of the reusable asset. In the usage part one shall find the set of activities that need to be carried out to reuse the artifact as was intended by the artifact producers. This can be as simple as saying “copy the xyz.exe file onto the root directory of your machine and run it, e.g. by double clicking on its icon or by entering C:/xyz.exe followed by a carriage return on a DOS command line”. But activities may also be much more specific and elaborate. For example, they may describe that one has to set the value of a specific static parameter at compile time to configure the error handling behavior of the component. In general, activities shall describe how to deal with every variability point of a reusable software asset, i.e. how the foreseen customization of the reusable asset is to be accomplished.

7.2.2 Quality Description Extensions to the RAS

In order to accommodate our quality attributes for software components we suggest to extend the RAS core metamodel as shown in Figure 7-5. Our extensions to the RAS core metamodel in Figure 7-5 are a direct consequence of Figure 7-2, which represented the metamodel for the quality description of software components (see also the Part III Overview Figure). Since RAS extensions are realized through profiles, our suggestion represents those key parts of a software component profile that are needed to cope with the description of quality attributes.

For the incorporation of our software component quality attributes we take two major modeling decisions, which amount to conceptual specializations of the RAS:

- We use the Classification section and its structural elements *Descriptor-Group* and *Descriptor Name-Value Pair* as metamodel concepts of our three quality sections and our quality attributes, respectively (see Figure 7-5).
- We use the concept *Artifact* with its characteristic of being *nested* (see also Figure 7-4) to model a software component and its syntactical elements (see Figure 7-5).

The UML Class diagram in Figure 7-5 depicts the original RAS core metamodel as metaclasses with no fill colors and associations between these metaclasses. Our profile concepts are represented as instances of the metaclasses with grayed out color. The key messages of the figure are paraphrased in the subsequent paragraph.

According to the RAS, the Classification part of a reusable asset is divided into *Descriptor-Groups*, which in turn contain semantically related *Descriptor Name-Value Pairs*. Each Descriptor-Group may conform to a *Classification Schema* that has been defined for some domain. The form and content of the classification schema is undefined. We consider Table 21 of Section 7.1.3 to be such a classification schema for the description of quality attributes of software components. The Descriptor-Groups correspond to our quality sections (Process Quality, Static Product Quality, and Quality of Service). A Descriptor Name-Value Pair corresponds to our notion of a Quality Attribute. Each Descriptor Name-Value Pair, and thus each Quality Attribute, conforms to a defined *Descriptor*. Through that conformance, every Quality Attribute is related to a *Descriptor Value*, defined by a Metric, and it is related to one or more *scopes* (of type *Artifact*). The *name* attribute of a Descriptor is formed by the combination “Characteristic:Quality Attribute” (see the respective columns in Table 21). The *value* attribute results from the informal definition in the Metric column of Table 21. The scope must be at least one of the defined syntactic elements of a software component, as used in Table 21. If we want to provide additional contextual information, for instance a usage pattern as mentioned in Section 7.1.2, we might do so by referring to further dedicated Artifacts (or scopes as seen from the Descriptor point of view). In the case of the usage pattern example, the Descriptor may refer to an Artifact being a UML sequence diagram.

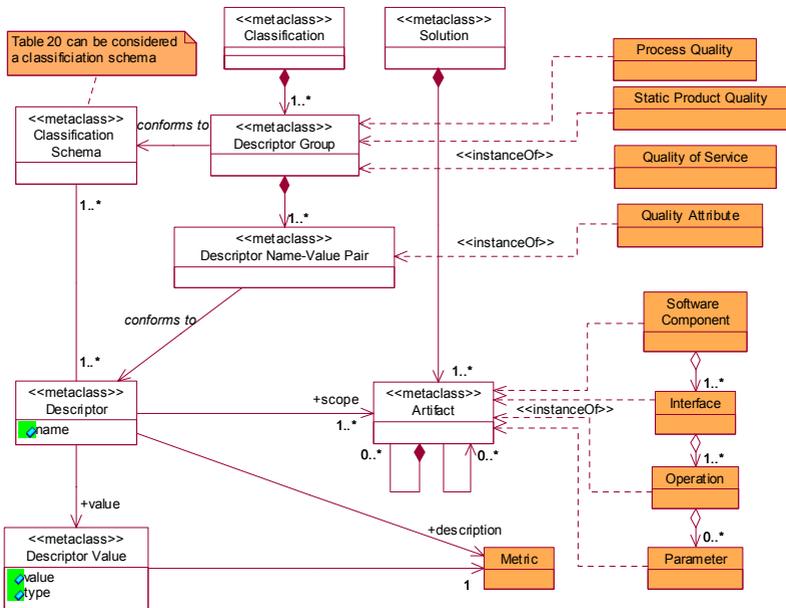


Figure 7-5: Key Extensions of RAS Concepts to Describe Quality Attributes of Software Components

7.2.2.1 A Simple Example

As a simple example of a RAS compliant quality attribute description, which follows the rules as defined in Figure 7-5, consider the following: An asset classification for a software component contains under the Descriptor-Group “Static Product Quality” the quality attributes “reusability–complexity:number of provided interfaces” and “reusability–complexity:parameter ratio”. They are listed in Table 21 and shown below again for convenience.

reusability - complexity	number of provided interfaces	quantitative measure which specifies the number of provided interfaces	Component
reusability - complexity	parameter ratio	quantitative measure which specifies the average number of parameters per operation (sum of parameters divided by sum of operations)	Component, interface

These two quality attributes shall be used to describe a Microsoft COM-compliant component. In this case, we picked a COM component that is even supplied by Microsoft: the MSXML, their component to programmatically access XML documents. We show an excerpt of the relevant parts of the IDL file below:

```

...
interface IXMLElementCollection : IDispatch
{
    [propput, id(DISPID_XMLELEMENTCOLLECTION_LENGTH)] HRESULT length([in] long v);
}
    
```

```

[proppget, id(DISPID_XMLELEMENTCOLLECTION_LENGTH)] HRESULT length([retval, out] long * p);
[proppget, restricted, hidden, id(DISPID_XMLELEMENTCOLLECTION_NEWENUM)] HRESULT _newEnum([retval, out] IUnknown **
ppUnk);
[id(DISPID_XMLELEMENTCOLLECTION_ITEM)] HRESULT item([optional, in] VARIANT var1,[optional, in] VARIANT var2,[retval,
out] IDispatch ** ppDisp);
};
...
interface IXMLDocument : IDispatch
{
    [proppget, id(DISPID_XMLDOCUMENT_ROOT)] HRESULT root ([retval, out] IXMLElement * * p);
    [proppget, id(DISPID_XMLDOCUMENT_FILESIZE)] HRESULT fileSize([retval, out] BSTR * p);
    [proppget, id(DISPID_XMLDOCUMENT_FILEMODIFIEDDATE)] HRESULT fileModifiedDate([retval, out] BSTR * p);
    [proppget, id(DISPID_XMLDOCUMENT_FILEUPDATEDDATE)] HRESULT fileUpdatedDate([retval, out] BSTR * p);
    [proppget, id(DISPID_XMLDOCUMENT_URL)] HRESULT URL([retval, out] BSTR * p);
    [propput, id(DISPID_XMLDOCUMENT_URL)] HRESULT URL([in] BSTR p);
    [proppget, id(DISPID_XMLDOCUMENT_MIMETYPE)] HRESULT mimeType([retval, out] BSTR * p);
    [proppget, id(DISPID_XMLDOCUMENT_READYSTATE)] HRESULT readyState([retval, out] long *p);
    [proppget, id(DISPID_XMLDOCUMENT_CHARSET)] HRESULT charset([retval, out]BSTR *p);
    [propput, id(DISPID_XMLDOCUMENT_CHARSET)] HRESULT charset([in]BSTR p);
    [proppget, id(DISPID_XMLDOCUMENT_VERSION)] HRESULT version([retval, out]BSTR *p);
    [proppget, id(DISPID_XMLDOCUMENT_DOCTYPE)] HRESULT doctype([retval, out]BSTR *p);
    [proppget, id(DISPID_XMLDOCUMENT_DTDURL)] HRESULT dtdURL([retval, out]BSTR *p);
    [id(DISPID_XMLDOCUMENT_CREATEELEMENT)] HRESULT createElement([in] VARIANT vType, [optional, in] VARIANT var1,
[retval, out] IXMLElement * * ppElem);
};
...
interface IXMLElement : IDispatch
{
    [proppget, id(DISPID_XMLELEMENT_TAGNAME)] HRESULT tagName([retval, out] BSTR * p);
    [propput, id(DISPID_XMLELEMENT_TAGNAME)] HRESULT tagName([in] BSTR p);
    [proppget, id(DISPID_XMLELEMENT_PARENT)] HRESULT parent([retval, out]IXMLElement **ppParent);
    [id(DISPID_XMLELEMENT_SETATTRIBUTE)] HRESULT setAttribute([in] BSTR strPropertyName,[in] VARIANT PropertyValue);
    [id(DISPID_XMLELEMENT_GETATTRIBUTE)] HRESULT getAttribute([in] BSTR strPropertyName,[retval, out] VARIANT*
PropertyValue);
    [id(DISPID_XMLELEMENT_REMOVEATTRIBUTE)] HRESULT removeAttribute([in] BSTR strPropertyName);
    [proppget, id(DISPID_XMLELEMENT_CHILDREN)] HRESULT children([retval, out] IXMLElementCollection * * pp);
    [proppget, id(DISPID_XMLELEMENT_TYPE)] HRESULT type([retval, out] long *p);
    [proppget, id(DISPID_XMLELEMENT_TEXT)] HRESULT text([retval, out] BSTR *p);
    [propput, id(DISPID_XMLELEMENT_TEXT)] HRESULT text([in] BSTR p);
    [id(DISPID_XMLELEMENT_ADDCHILD)] HRESULT addChild([in] IXMLElement *pChildElem, long lIndex, long lReserved); //
lReserved must be -1
    [id(DISPID_XMLELEMENT_REMOVECHILD)] HRESULT removeChild([in]IXMLElement *pChildElem);
};
...
interface IXMLError : IUnknown
{
    HRESULT GetErrorInfo(XML_ERROR *pErrorReturn);
};
...
interface IXMLElementNotificationSink : IDispatch
{
    [id(DISPID_XMLNOTIFSINK_CHILDADED)] HRESULT ChildAdded([in]IDispatch *pChildElem); // If input param is NULL then
parse of this elem is done
};

```

The number of provided interfaces is *five* and the average number of parameters per operation is *1,25* (40 parameters over 32 operations). The Quality Attributes would therefore be:

reusability–complexity: number of provided interfaces = 5

reusability–complexity:parameter ratio = 1.25

The quality attributes are conformant with the Descriptor definition in Figure 7-5:

<p>name: reusability–complexity:number of provided interfaces</p> <p>value: number of provided interfaces</p> <p>type: integer number</p> <p>scope: MSXML</p> <p>description: quantitative measure which specifies the number of provided interfaces</p>	<p>name: reusability–complexity:parameter ratio</p> <p>value: average number of parameters per operation</p> <p>type: real number</p> <p>scope: MSXML</p> <p>description: quantitative measure which specifies the average number of parameters per operation (sum of parameters divided by sum of operations)</p>
---	---

7.2.3 Proposal for a RAS Physical Model

The RAS physical model defines how the constituents of an asset package, which logically belong together, are physically arranged in concrete files and directories. The RAS is not prescriptive or normative here, and allows basically any form of physical structure. XML [23] is indicated as a preferred technology.

In order to define description templates for defined types of assets, one has to develop asset-type specific profiles (recall Figure 7-3). The RAS provides a Microsoft Word® document to describe RAS profiles in an informal way. However, the RAS specifies neither the physical representation of RAS profiles nor how an instance file for the description of an individual software asset has to look like. Hence, the machine-processable format is not normatively defined in the RAS.

Being involved in an enterprise-wide standardization effort for describing reusable software assets [109], we found that an agreed physical representation of the description of software assets is absolutely required. Our direct or indirect requirements on such a physical representation are the following:

- R1. It must be amenable to tool processing, because we want to automate the handling of software assets, along the entire reuse process, by a tool infrastructure.
- R2. The tool infrastructure shall be independent of the type of software assets processed. I.e. new types of software assets need to be introduced without any change in the tool landscape.
- R3. Producers of reusable assets shall be forced to include all required information in an asset description, but they shall be allowed to include more, if they want, in a compliant way. Thus, respective rules and constraints shall be automatically checkable and enforceable with a validation step.
- R4. The development of proprietary validation software shall be avoided as far as possible, i.e. standard technology shall be used, where parsers, validation software etc. are readily available.
- R5. Software asset descriptions shall remain human read- and interpretable.

In an envisioned XML-based approach to physically represent both RAS profiles and individual software asset descriptions, each RAS profile would be represented as an XML schema following the XML Schema recommendation [44], and each description of an individual software asset would be represented as an XML document. The physical layer of Figure 7-3 shows this relationship. The profile-specific XML schemas we call *RAS profile XML schema* and the XML documents, instantiated from a RAS profile XML schema, we call *RAS XML document*. While the RAS includes a representation of the RAS core metamodel as an XML Schema file (the *RAS core XML schema*), it does *not* say how the RAS profile XML schema is derived from the RAS core XML schema or how it formally relates to it. It is therefore questionable if the RAS core XML schema serves more than as a starting file to edit ones own profile by replacing/adding/changing XML elements and XML attributes as needed. Consequently, it is also unclear what kind of formal relationships, if any at all, there are between the RAS core XML schema and a RAS profile XML schema as indicated by the “?” in Figure 7-3.

However, if we assume such an XML-based approach, the RAS profile XML schema can be used for the validation of RAS XML documents, as well as for the tool-supported processing of these asset descriptions along the reuse process. It is therefore amenable to some of the aforementioned requirements. Once we assume an XML-based approach, the previously listed general requirements transform into specific requirements:

- R6. All constraints and rules (as discussed under item R3 above) have to be expressed in the RAS profile XML schema, which then enables automatic validation and the usage of commercially available XML validation software.

R7. Relationships to the RAS core metamodel concepts need to be expressed in the RAS profile XML schemas and RAS XML documents for the below reasons:

- We need to be able to validate the introduction of new types of software assets, in the form of new RAS profile XML schemas, against the RAS core metamodel without modification of the existing tool infrastructure (derived from R2).
- We need to be able to accept (i.e. to validate) elements and attributes that are provided by the producer but are exceeding the required minimal information (derived from R3),
- We want humans to be able to interpret RAS XML documents (derived from R5). For example, they shall be able to understand that when they find “development-environment” in the RAS XML document this is meant to be a specific “classification descriptor” (a core metamodel concept) of the asset at hand.

In the process of creating our own, company- and application domain specific RAS profile XML schemas, we were confronted with the problem of fulfilling our requirements listed above. The RAS, at its current stage, is too vague and informal in specifying how one should create own RAS profiles and how to best design a physical representation for them that incorporates content constraints and metamodel information. Specifically, the provided example of the “Any” profile, which is part of the latest RAS release, is not helpful in that respect, because it is open to interpretation.

In Appendix B, we provide alternatives for a RAS profile XML schema with weaker or stronger relationships to the RAS core XML schema. The alternatives provide gradual improvements over the basic variant discussed in the RAS. The improvements of the discussed alternatives focus particularly on fulfilling the requirements R2, R5 and R7. As a consequence, these alternatives include metamodel information in the RAS profile XML schema and thus also in the RAS XML documents. The metamodel information can be expressed through XML constructs such as (a) XML element names (discussed in Section B 2), (b) XML attributes (B 3), (c) XML types (B 4), or (d) through type hierarchies by means of XML complex types (B 5). As a result of the evaluation of these alternatives and as argued in Appendix B, we have chosen the variant in which the references to the metamodel are included as XML attributes.

7.3 ABB RAS Profiles

As a practical example, in which we have used the idea of a software component data sheet and the RAS, we will briefly mention the ABB RAS profiles. We have created them to describe reusable application components in a standardized way.

ABB has recently introduced an automation platform for process control – the Aspect Integrator Platform (AIP). The platform is applied in several vertical market domains, i.e. it is used for the control of chemical processes, of oil and gas refineries, of manufacturing lines, of power systems grids, etc. Applying means to develop market specific applications on top of the platform and tailor such applications to needs of specific customer orders.

Because of the large potential for reusing the platform and for reusing application components within and across market domains, we investigated the feasibility of defining an ABB-wide systematic reuse process [108], and as part of this activity to define a global standard for describing reusable asset packages related to the AIP.

On a high-level, the AIP paradigm is one where applications are built from objects. So one could say it is an object-oriented paradigm. However, the notion of object and application is not to be understood on the low level of object-oriented programming language concepts. It is rather suited to the abstraction level of an application engineer (a chemical engineer, an electrical engineer, etc.). That is, an object refers to a semantically rich domain object (e.g. a valve, a pipeline, a belt, a power transformer, etc.) that has rich behavior and a diversity of relevant data associated with it. The AIP has

therefore introduced the nomenclature *AspectSystem* and *AspectObject* to stand for application and object, respectively. Because of the diverse kinds of applications normally realized by an industrial automation system, *AspectObjects* group their support for specific applications into so-called *Aspects*. An *AspectSystem* is therefore using certain *Aspects* of an *AspectObject* to realize its functionality. As an illustrative example consider the informal drawing in Figure 7-6. It shows a real object: a valve. This object is conceptualized as an *AspectObject* in the AIP. This valve-type *AspectObject* provides several *Aspects*: a *Control Aspect* that encapsulates the customizable valve control logic, a *Faceplate Aspect* that provides a GUI element to supervise and control a valve from an operator station, etc. Specific applications, such as a *Control Builder* tool or an *Active Graphics* application, are using the *AspectObjects* and their provided *Aspects*. For instance, the *Control Builder*, an engineering tool, allows designing in a graphical way the control logic of a valve according to customer needs. An *Active Graphics* application will use the *Faceplate Aspect* of an *AspectObject* to create process control screens that allow an operator to remotely supervise the status of a valve as well as remotely control it.

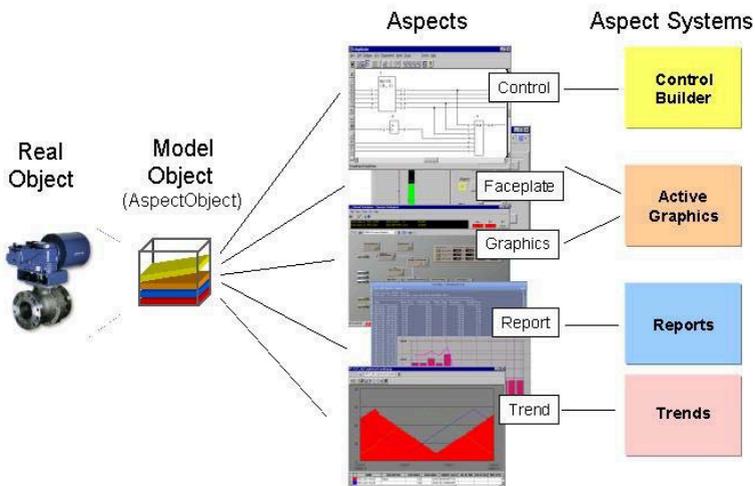


Figure 7-6: The AIP Application Paradigm

Already from this example one can infer that the reuse of well-developed types of *AspectObjects* and well-developed types of *Aspects* might be beneficial. ABB application engineers around the globe could for instance reuse the *AspectObject* of a specific type of valve in all their customer deliveries that include such a specific type.

Our choice of reusable asset types related to the AIP can be motivated by the conceptual model of Figure 7-7, which shows the static relationships among major programming and deployment concepts of an AIP-based application.

Building upon our discussion on *AspectObjects* and *Aspects* above, the respective type definitions of these two major concepts are the so-called *ObjectTypes* and *AspectTypes*. The actual implementation of an *AspectType* is realized by a set of *AspectSystemObjects*. An *AspectSystemObject* is a Microsoft COM component that adheres to special rules defined by the AIP.

These rules essentially amount to the provision of some AIP-specific, predefined COM interfaces. Note that the discussed conceptual model is somewhat simplified and does not show some of the intricacies. However, it is conceptually correct at the chosen level of abstraction and suits the purpose of our discussion. A more detailed discussion of both the AIP and the reusable assets can be found in [109].

As the main reusable assets of interest we have identified and created RAS profile XML schemas for: (a) AspectSystems, as entire “applications” that contain Aspect Types which are potentially applicable to different types of ObjectTypes, (b) ObjectTypes (or ObjectType libraries), and (c) AspectSystemObjects (i.e. individual COM software components).

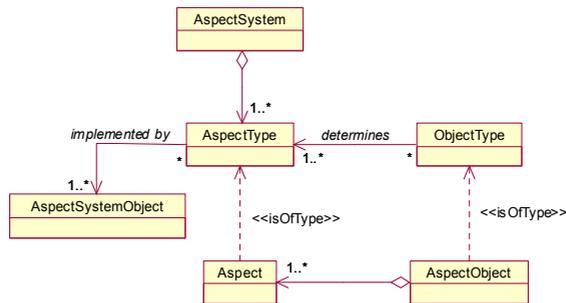


Figure 7-7: Main Application Programming Concepts in the AIP

The RAS profiles for these individual asset types were created according to our proposal for a physical model as detailed in Section B 3. The usefulness of the physical model in the context of these profiles has been explored through the development of an asset package viewer [139], which does not need any profile knowledge. Since both the details of the profile schemas and the viewer are no special research contribution, we will not elaborate further on them in the scope of this thesis report.

However, we would like to report on some of our experiences with regard to asset descriptions and the use of the RAS in the context of the AIP:

- The lack of a standardized RAS physical model and the lack of clear rules for physical models of profiles could largely reduce the usefulness and therefore the adoption of the RAS in general. We have experienced that one can spend much time in defining a physical model because there are many options. If we fail to have a standardized physical model we will not have the benefits through generically applicable tool automation such as asset packagers, asset repositories, publishing tools, search facilities, etc. The RAS would be reduced to being a recommendation for what should “somehow be documented” by a reusable asset provider.
- The terminology of the RAS (such as asset, artifact, variability point, variability-point binding, context, context-free binding, etc.) must be hidden from the users of assets. By users we mean both people engaged in the production and description of assets and people engaged in the search for and consumption of assets. These terms, which are generic concepts anyway, were reported to be confusing for practicing engineers. As we experienced, they may represent a barrier to adopting the practice of describing reusable assets. Developers and users do not want to spend the time to find out what a particular general term could mean. If it is not clear right away, they will simply skip it.

- The time needed to elicit the value of descriptors is inversely proportional to the willingness of asset developers to provide descriptors. Automatic retrieval of asset descriptor values is therefore highly recommended. In the AIP context, we started a project to explore the extent with which the various properties can be automatically or semi-automatically obtained and packaged by an asset packager. This is clearly something that can only be done in a constrained and controllable environment. While the problem of automatically deriving values is a general one, will the AIP-based solution hardly have universal applicability.
- Descriptor-groups and descriptor names are domain specific. It is therefore of little value to try and find *the* (universally) *right* classification. A pragmatic approach is to be favored. This was one reason why we tried to understand how a customized set of descriptors is found in a guided way and why we came up with the constructive quality model as discussed in Section 6.2.
- Our first versions for the three AIP-related reusable asset profiles contain only a very small number of quality attributes. In fact, as indicated by the domain specific nature of classifications, they are sometimes not even explicit descriptors but indirectly derived information. For instance, the AIP defines different basic types of AspectSystemObjects (ASO). One of them is a so-called DataASO. Being of such a type means, among other things, to support some form of persistence. Hence, providing a classification descriptor indicating the type of ASO and possibly its specific means of supporting persistence is, in the constrained reuse context of the AIP, more informative than providing a quality attribute such as “reliability–recoverability:persistence” (Table 21). Again, this observation confirms the higher importance of a systematic approach to constructing a quality description over an endless (re-)search for the single right one.
- (Re-)users are hardly ever searching for one single software component. Our experience in the AIP context showed that (re-)users are interested in a partial solution, which for them is a piece of functionality. Hardly ever does this map to one software component in the technology sense of the word. It rather requires component systems in the form of coherent sets of useful business objects and infrastructure objects. In the AIP, these are the ObjectTypes and AspectSystems, respectively. Only these conceptual components (or conceptual systems, whatever the viewpoint) are artifacts useful for reuse. Their value lies in the large domain effort and expertise that has gone into their development and to a lesser extent in the technology related effort (i.e. in the implementation effort to bring them into technological software components). However, technological software components (in the AIP the AspectSystemObjects realized as a COM components) provide a valuable packaging entity for realizing and deploying conceptual components.

7.3.1 Enforcing AIP-Related Asset Descriptions

For any implementation of a process within a company, especially one that involves humans and implies change, there is a high degree of management commitment and attention required. This is no different and even of utmost importance for a systematic reuse process.

As part of the AIP-based application development process, we have identified the need for describing reusable assets. Hence, it must not only be ensured that reusable assets are of high quality but also that asset descriptions are so. In our case, we want to ensure the availability of minimal asset information through the aforementioned profiles and the fact that every AIP-related, reusable asset comes with a description that is compliant with such a profile. Within ABB we have established a certification program – the Industrial^{IT} Certification - that ensures conformance of the diverse ABB products with corporate standards. It was one of the top priorities and high on the management agenda over the last two years and still is. Industrial^{IT} refers to the initiative by which all products that are (even remotely) related to AIP-based systems must satisfy some minimal requirements. It was our

intend to include the requirements for the availability of RAS-compliant asset descriptions as one criteria into the certification schema. This proposal is still discussed but low in priority due to the fact that almost no Level 2 certified products (discussed below) exist.

Industrial^{IT} conformance is defined on four levels:

- Level 0. *Information*: A product (such as a motor, a switch, a controller, a software package, etc.), which is certified for level 0, is providing a defined set of documents in a defined electronic format. This shall simply replace bookshelves and binders of paper-based documents. The electronic format is such that the documentation can be read by an AIP-based system. As an example consider the valve again: The maintenance documentation of a valve must be produced in such a way that it can be accessed by an ABB-wide documentation AspectSystem. Level 0 thus constitutes a declaration of standardized product information, their formats, and procedures for easier engineering, operation, and maintenance.
- Level 1. *Connectivity*: A product, which is certified for level 1, can be physically connected, installed, operated, reinstalled and disconnected to/from an AIP-based system. The product will exchange information with the AIP-based system via defined interfaces, and basic data can be handled reliably over at least one defined communications protocol. For example, an SAP/R3 product planning application is level 1 certified because there is a wrapper that enables its exchange of planning files with an AIP-based system (e.g. a batch-process production system for chocolate products).
- Level 2. *Integration*: A product, which is certified for level 2, provides full integration into the AIP. That is, it provides its own AspectsSystems, AspectObjects, etc. User interfaces for navigation, distribution and presentation of information is based on the ABB Aspect Object technology and its user interface guidelines.
- Level 3. *Optimization*: A product, which is certified for level 3, demonstrates the use of the full scope of AIP-based features. It can make use of advanced functionality such as providing native language support, supporting single point configuration data entry, supporting complex object copy/paste (“deep copies”), supporting integration into ABB’s engineering tool frameworks, etc.

The AIP-specific RAS profiles are currently considered for inclusion into the level 2 (and level 3) certification. Level 2 would then require that

- the certification candidate, an AIP-related reusable asset, must be packaged and described according to its predefined profile,
- that exemplars of such asset descriptions have been verified and validated against such a profile, and
- that the quality and credibility of the description contents has been verified by the certification authority or some other party.

With the introduction of the defined profiles into the certification scheme, the associated tools such as asset packager and viewer would become integral part of the reuse process.

Summary of Part III

Part III presented a construction approach to create customized quality models. The approach responds to the need of adaptable property models and it takes into account the multi-faceted nature of properties, i.e. that the context of a property defines its meaning and utility. In our approach, the context is made explicit and in essence defined through stakeholders, taxonomic decompositions of high-level properties, life-cycle phases and distinct components as part of these life-cycle phases. Once the context is explicit, individual quality-carrying properties can be elicited by means of the GQM or through scenario-based approaches in general.

Part III further presented a structure for describing quality attributes of software components. Besides suggesting many non-functional quality attributes for software components, Part III showed how such descriptions can be represented in RAS profiles through defined descriptors. Because the RAS is not explicit about the electronic format of RAS profiles or RAS-compliant descriptions of individual assets, Part III (with the help of Appendix B) suggested an XML-based approach that satisfies several important requirements, in particular with respect to suitability for tool-based validation. The requirements for the RAS profiles and their concrete applications were discussed in the context of an ABB initiative to define standardized reusable assets related to an industrial automation platform.

8 Conclusion

Objectives: This Chapter summarizes the results reported in this thesis report and suggests future work based on the knowledge gained from our research and based on the status of the implementation or application of that knowledge to (software) engineering practice.

After having read this chapter the reader will:

- **know that we envision follow-on research activities in four different directions:**
 - (1) **multi-system and multi-property modeling based on the UML 2.0,**
 - (2) **metrics for models and their representations,**
 - (3) **experience gain with quality attribute descriptions of reusable assets**
 - (4) **enhancement of existing or new development methods with quality attribute modeling.**

8.1 Summary of Results

This thesis concentrated on generating a body of knowledge in support of the conception of basic methodological building blocks and fundamental underpinnings that would allow development methods and their modeling paradigms to eventually cope with the greater diversity of systems and greater diversity of properties. To this end, we developed a general philosophy of properties and systems that, through its generality, can serve as a conceptual aid to tackle some salient problems in conceptualizing non-functional properties of/and hierarchical systems, but that can also serve as a generalization and means to explain and relate existing works.

Our general philosophy yielded two types of results: basic tenets and a (meta) model of systems and properties. The tenets are meant to provide a conceptual aid that focuses on shaping an attitude of mind that is helpful to understand, explain, and approach the modeling of properties in a holistic way. The (meta) model of systems and properties distinguishes between the two basic concepts - component and process - and it distinguishes among four fundamentally different types of properties of concrete components: physical-, functional-, extra-functional-, and conceptual properties. Each of these properties is different in nature and requires a different approach to be elicited, modeled, and realized. Systems are special types of components in that their internals are of interest to us. The distinction between concrete and conceptual systems and the explication of their respective suprasystems allows us to systematically discover properties that belong to these four basic types.

Following the systems inquiry paradigm, the application of our philosophy, in the form of knowledge-based actions, involves two stages. In the first stage, we have suggested general conceptual building blocks that might be incorporated in systems development methods to more systematically cope with properties. This includes the 2-2-2 model applied to discovering stakeholders, through which the relevant properties can be identified by means of the GQM or scenarios. The 2-2-2 model is an application of our tenets. This further includes a principled way by which property realizations can be traced through hierarchies of systems. In order to accomplish property traceability in a model-based way, among others to facilitate tool support, we showed how the UML V1.40/1.50 could be extended through its lightweight extension mechanisms. Through our choice of modeling systems as UML Subsystems, we chose UML TaggedValues as the UML representation for all but functional properties and we chose stereotypes of UML Abstractions to represent semantically meaningful property traces. Property traceability and its representation is an application of the meta-model of systems and properties.

In the second stage of application of our philosophy, this thesis proposes an approach to construct customized quality models. Given the huge variety of systems (or components in general) to which we want to ascribe properties, no single quality model can do. Our construction approach is flexible in that it is based on four dimensions, each dimension representing a customizable set: life-cycle phases, components within phases, stakeholders, and taxonomies of quality attributes per stakeholder. The choice of one element {life-cycle phase, component, stakeholder, taxonomy leaf-node} represents a well-scoped context to define measurable properties for the component. The discovery of these properties is preferably done by means of the GQM or scenarios.

The second stage also suggests a structure for describing quality attributes of software components as reusable assets. It also suggests concrete quality attributes for software components. The structure is a result of both the tenets and the customizable quality model. In order for such quality descriptions to be standardized and machine-processable, we propose to use the RAS and we therefore suggest and evaluate possible XML-based alternatives to represent RAS profiles in an electronic format. Finally, we briefly show how we applied the idea of reusable assets and the application of the RAS to describe reuse artifacts in the context of ABB's Aspect Integrator Platform.

8.2 Future Work

In this Section we present further work as possible continuations of our activities. These activities are motivated by either the limitations of our work or by further questions that came up during our work but where deeper exploration would have exploded the scope of this PhD. I personally hope that some of the suggested activities are pursued, especially because they would make some of our more theoretical work more tangible and applicable to practicing engineers.

8.2.1 UML Extensions Necessitating More Than Lightweight Profiles

As we have indicated in Section 5.3, we had to bend the UML as defined in Version 1.4 or 1.5 to satisfy our systems modeling needs. In our view, the lightweight, profile-based extensions that we proposed are pragmatic solutions to benefit from current toolsets. However, we would require some basic features in the UML, which are currently not taken care of. We have therefore listed them in the future work Section of this report as guidance for further investigations. At the time of this writing the OMG has just released the UML 2.0. Our preliminary study of the one submission [133] (aka U2P proposal), which was chosen to become the UML 2.0, indicated that several basic requirements we have identified below can potentially be met in a more elegant and formally satisfying way. We have therefore briefly indicated our main requirements and a conceivable solution approach based on the U2P proposal.

The following general shortcomings are found in UML 1.4 (and 1.5) for the purpose of modeling of properties and/of general systems:

a) Hierarchical Classifier or in general composite structures

UML 1.4: The UML meta-class Classifier is not defined as being hierarchic. In fact, hierarchy (or level) is not an explicit concept in the UML. Some subtypes of Classifier have defined some special kinds of hierarchies in non-inform ways (like nested Classes, or Subsystem may reference realization elements). However, this is not general and not flexible enough. Since whole-part decompositions, with the characteristic of encapsulating/abstracting the parts, are the most natural way to think, almost every modeler is forced to find her own way of representing hierarchies of things of interest. Recently researchers provided suggestions to more formally introduce semantically varying part-whole relationships in the UML metamodel [12].

UML 2.0: The U2P proposal defines in its superstructure a framework for Composite Structures (Chapter 3 in [133]), which allow to model internal structures of Classifiers and also internal behavior based on state machines (discussed in Chapter 9 of the U2P proposal).

b) Property

UML 1.4: The UML semantics (Part 2) does not define an explicit concept “property”, one that exceeded the very narrow meaning of a behavioralFeature (Method or Operation) or structuralFeature (Attribute) of a Classifier. Nonetheless, the UML 1.4 uses the term ‘element property’ but only as a concept in its graphical notation (Part 3, graphical representation) that lacks a counterpart the UML semantics convention (Part 2).

UML 2.0: The U2P proposal defines in its infrastructure proposal an explicit meta-class *Property*, which refers to structural features. Property is a new, additional concept to StructuralFeature as defined in the UML V1.4. The usefulness of Property should therefore be explored in detail.

c) Parametric Property

UML 1.4: UML does not support the notion of parametric properties (parametric UML Features of a Classifier) or parametric relationships (parametric UML MappingExpression of an Abstraction). Especially, continuous time varying properties for system engineering, but also any parametric quality attribute would need such a construct. For example, “response-time(CPU-speed)” would identify the extra-functional property response-time to be parametric, i.e. dependent on the CPU-speed. Parametric properties would make it easier to define constraints for attributes. Introducing stereotyped methods (where the stereotype semantics defines them as non-callable) is a plausible workaround at least for parametric properties [29].

UML 2.0: No explicit support found in U2P.

d) Notion of Concrete System

UML 1.4: The notion of a concrete system that satisfies our criteria defined in Section 5.3.2.1, is not an explicit part of the UML. Even the concept of a system is not explicitly defined. The UML itself uses the term physical system in descriptive, informal passages when the semantics of model elements is explained. This shows that a distinction between the model of a system and the system itself would be appropriate. Some tool vendors introduced <<system>> as a stereotype of Package for model organization purpose. The UML defines <<systemModel>> as a defined stereotype of Model. Again, it serves the model organization only.

UML 2.0: The U2P proposal for a Component and in particular for complexComponent should suffice to represent our main concerns for concrete system. See also comments made in a) above.

e) More Explicit Support for Matter/Energy/Information Flow

UML 1.4: Several sources have argued for more explicit support for data flow, and in general information flow modeling. Data flow models/diagrams are still widely used in practice, yet the UML does not provide the basic constructs and thus of course also no explicit data flow diagrams. Depicting Matter/Energy/Information flow is also crucial for systems engineering. Our current workaround is to show Matter/Energy/Information flow as object flow in UML Activity diagrams. This was for instance used in Figure 4-11 on page 78.

UML 2.0: The U2P proposal introduces both an explicit superstructure package called InformationFlow and extensions to ActivityGraphs, which stem from the rich definition of new

concepts in newly defined Activity and Action packages (U2P proposal Chapter 6 and Chapter 5, respectively).

8.2.2 Modeling Metrics

An entire research topic of its own would be the definition of properties of conceptual systems, such as of model concepts and of model representations.

In our thesis, we have concentrated mainly on properties of concrete components and on properties of conceptual components in the context of the behavior template as a special kind of information. In Section 4.1, though, we had intuitively argued that a good model and its associated representation is one that, besides being widely agreed upon, can be efficiently processed by humans, can be efficiently processed by computers, and that serves the purpose of the model in the first place.

It would be very interesting to investigate which properties of a conceptual construct and its representation make it amenable for efficient, and purpose-oriented human processing as well as computer processing. For instance,

- *Meta-models and human processing*: to what extent do meta-concepts facilitate human processing of a model
- *Metrics for models*: which inherent properties do conceptualizations have and how can they be measured and correlated to human- and computer processing? To what extent does the number and richness of concepts and concept relationship support or prevent human- and computer processing?
- *Metrics for model representations*: which inherent properties do model representations have, how can they be measured, and how can they be correlated to human- and computer processability?
- *Executable models*: which human-processable representations are amendable also for creating machine executable models?
- *Friction points*: Which properties are the cause for the dichotomy between human-optimized and computer-optimized models and their representations?

Answers to these kinds of questions provide valuable input to the design of future modeling languages and their representations.

8.2.3 Gaining Experience with Quality Attribute Descriptions of Reusable Assets

There is little practical experience reported in current literature about the effort needed to describe reusable assets that goes beyond what one would normally do for single development efforts, about the extent with which reusable assets are described today, and how useful (what kinds of) descriptions of reusable assets are for consumers. All these aspects would of course be particularly interesting with respect to quality attributes. While one can define many metrics for attributes of reusable assets, all of them certainly having some meaningfulness, it would be interesting to explore which ones are really considered during evaluation and application of a reusable asset and which ones have a pure academic character only.

We would therefore greatly support a research effort that would predominantly collect such information and would try to structure and generalize it in order to make statements about the level of description that is feasible and practical. We hope that the practical experience eventually gained by applying the RAS profiles could indicate some directions. But because such a research effort would mainly be conducted with companies that practice systematic reuse, and because there are not that many which do so successfully, this task is a formidable one.

8.2.4 Enhancement of Development Methods and Tool Support

The original, practical motivation of our work related to the immaturity of current systems and software development methods with respect to their support for hierarchical systems modeling and with respect to their support for non-functional properties.

In this thesis we have provided some conceptual building blocks of a model-based, multi-system multi-property development method, which we would now like to incorporate into existing development methods or even develop our own method, if so needed. With respect to the introduced methods in Section 2.4 and their enhancement with our results, we would envision the following research work as indicated in Table 22.

Table 22: Potential Future Work: Combining Results of the Thesis Work with Existing Development Methods

Development method	Type of investigation
OOSEM	<ul style="list-style-type: none"> • Focus on the integration of UML-based property traceability (as discussed in Section 5.3). This would require an explicit meta-model of the organization of a system description in the OOSEM (similar to our Figure 5-9), as well as the reconsideration of the UML Class concept to represent a system. • Exploration with the property-related integration of the OOSEM's Requirements and Verification Traceability database (RVT) into OOSEM's system models.
OPM	<ul style="list-style-type: none"> • Because OPM's meta-model is close to ours (i.e. close to Figure 4-8) but OPM focuses on functional properties only, the primary focus should be on the exploration of OPM's enhancement to also handle non-functional properties (mainly extra-functional properties) and support property traceability. • Although not an intention of the author's of the OPM, the use of the UML/OCL could be explored (along the line of our Section 5.3.2). Because OPM introduces a small number of concepts only, this would result in a small, over-seeable version of the UML. Also, OPM's attractiveness for greater industry adoption would improve.
Catalysis	<ul style="list-style-type: none"> • Because Catalysis is focused on information processing, the first immediate interesting aspect would be to explore the possibility of supporting extra-functional properties. Together with the inclusion of extra-functional properties the support for property traceability can be investigated as an enhancement of the current Catalysis traceability support (tracing of collaboration and component refinements). • In order for Catalysis to serve as a systems development method, major investigations with resulting possible changes to the meaning and definition of the current concepts would be necessary (e.g. to support concrete systems with matter/energy processing). We would therefore probably not attempt to reuse or modify Catalysis to serve such a purpose.
UML Components	<ul style="list-style-type: none"> • UML Components is the one method that does not provide any of the main features required to make it a systems development method. Similar to Catalysis but even more pronounced, UML Components focuses on the modeling of information processing by means of software components. The envisioned steps of enhancements based on our works could be: (1) create a meta-model of UML Components' current notion of a system, (2) add to it the capability of modeling hierarchical systems, (3) support property traceability of functional properties, (4) add the handling of extra-functional properties (incl. traceability).

- KobrA
 - The same types of investigations and comments apply as for Catalysis. However, KobrA has advantages over Catalysis because of its more flexible and general approach on modeling hierarchical systems. Further, some afunctional properties are addressed through a suggested development process.
- ABD
 - As we mentioned in Section 2.4, ABD is not a full-fledged development method. With regard to our work it can be considered an approach that addresses the logical decomposition of a system (cf. Figure 4-14 on page 83). Since the ABD does not explicitly address the traceability issue but does cope with several basic types of properties, it could benefit most from the inclusion of the traceability approach for functional, extra-functional and afunctional properties.

In order to better understand how to operationalize our methodological building blocks, we have recently started to explore with our own method. But because the Systemic Enterprise Architecture Methodology (SEAM [140]) and our work have their roots at the same university laboratory, there are natural connections and a common attitude of mind. This suggests looking into combining those aspects of our work that relate to systems and properties modeling with the SEAM approach. By this, we hope to be able to validate and refine some of our more theoretical results by means of a method that is applied to a successive number of cases.

It goes without words that tool support for the modeling of systems and properties is indispensable. Section 8.2.1 indicated what kind of support is required from modeling notations. But this is only one step. The next level of support must be that tools verify and enforce property traces and also help to navigate along property traces or to identify property influences. For instance, with respect to verifying property traces one would expect that if a functional property is ascribed to a system, the tool should enforce that the modeler has created a realizing collaboration or that she explicitly denies doing so for good reasons.

Bibliographic References

- [1] J. O. Aagedal, Quality of Service Support in Development of Distributed Systems, PhD Thesis, University of Oslo, Department of Informatics, Oslo, 2001
- [2] J. O. Aagedal and A.-J. Berre, "ODP-based QoS-support in UML," *IEEE Software*, vol. 8, pp. 310 - 321, 1997.
- [3] Altova. (2002). XML Spy. Altova GmbH & Altova Inc. [Online]. Available: www.xmlspy.com
- [4] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wust, and J. Zettel, *Component-Based Product Line Engineering with UML*, 1st ed: Addison-Wesley, 2001.
- [5] C. Atkinson, B. Paech, J. Reinhold, and T. Sander, "Developing and Applying Component-Based Model-Driven Architectures in Kobra," in *Proc. Fifth IEEE International Enterprise Distributed Object Computing Conference*, 2001, pp. 212-223.
- [6] R. Audi, *The Cambridge Dictionary of Philosophy*, 2nd ed. Cambridge: Cambridge University Press, 1999.
- [7] D. Auf-der-Maur, O. Preiss, T. Siegrist, and A. Wegmann, "CBSE and embedded systems - Do they match?," in *Proc. Workshop on Pervasive Component Systems as part of ECOOP 2000*, 2000.
- [8] D. E. Avison and G. Fitzgerald, "Where Now for Development Methodologies," *Communications of the ACM*, vol. 46, 2003.
- [9] F. Bachmann, L. Bass, G. Chastek, P. Donohoe, and F. Peruzzi, "The Architecture Based Design Method." Carnegie Mellon University - Software Engineering Institute, Pittsburgh, Technical Report CMU/SEI-2000-TR-0012000.
- [10] B. Banathy. (2001, March). A Taste of Systemics. The Primer Project, A Special Integration Group of the International Society for the Systems Sciences (ISSS) [Online]. Available: <http://www.iss.org/taste.html>
- [11] J. Bansiya and C. G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment," *IEEE Transactions on Software Engineering*, vol. 28, pp. 4-17, 2002.
- [12] F. Barbier, B. Henderson-Sellers, A. L. Parc-Lacayrelle, and J.-M. Bruel, "Formalization of the Whole-Part Relationship in the Unified Modeling Language," *IEEE Transactions on Software Engineering*, vol. 29, 2003.
- [13] V. Basili and D. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering*, vol. 10, pp. 728-738, 1984.
- [14] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 6 ed. Reading, Massachusetts: Addison-Wesley, 1999.
- [15] L. v. Bertalanffy, *General System Theory: Foundations, Development, Applications*. New York: George Braziller, 1969.
- [16] M. F. Bertoa, J. M. Troya, and A. Vallecillo, "A Survey on the Quality Information Provided by Software Component Vendors," in *Proc. QUAOOSE Workshop at ECOOP'2003*, 2003.
- [17] M. F. Bertoa and A. Vallecillo, "Quality Attributes for COTS Components," in *Proc. 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2002)*, 2002.
- [18] A. Beugnard, J.-M. Jezeguel, N. Plouzeau, and D. Watkins, "Making Components Contract Aware," *IEEE Computer*, vol. 32, pp. 38-45, July 1999.
- [19] B. W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, vol. 21, pp. 61-72, 1988.

- [20] B. W. Boehm, P. Bose, E. Horowitz, and M.-J. Lee, "Software Requirements as Negotiated Win Conditions," in *Proc. International Conference on Requirements Engineering (ICRE)*, 1994, pp. 74-83.
- [21] B. W. Boehm, J. R. Brown, J. R. Kaspar, M. Lipow, G. McLeod, and M. Merritt, *Characteristics of Software Quality*. Amsterdam: North Holland, 1978.
- [22] S. Bot, C.-H. Lung, and M. Farrell, "A Stakeholder-Centric Software Architecture Analysis Approach," in *Proc. 2nd International Software Architecture Workshop (ISAW-2) as part of SIGSOFT '96*, 1996, pp. 152 - 154.
- [23] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler, "Extensible Markup Language (XML) 1.0," World Wide Web Consortium (W3C), W3C Recommendation, 6 October 2000.
- [24] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture*. Chichester, UK: John Wiley and Sons, 1996.
- [25] R. H. Campbell and A. N. Haberman, "The Specification of Process Synchronization by Path Expression," in *Proc. International Symposium on Operating Systems*, 1973, pp. 89-102.
- [26] M. Cantor, "RUP SE: The Rational Unified Process for Systems Engineering," *the Rational edge*, November 2001.
- [27] J. A. Carvalho, "Strategies to Deal with Complexity in Information Systems Development," in *Proc. 6th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2002)*, 2002.
- [28] P. Checkland and S. Holwell, *Information, Systems and Information Systems - making sense of the field*. Chichester, UK: John Wiley & Sons, 1998.
- [29] J. Cheesman and J. Daniels, *UML Components - A Simple Process for Specifying Component-Based Software*. New York: Addison-Wesley, 2001.
- [30] P. Clements and L. Northtop, *Software Product Lines: Practices and Patterns*. Boston: Addison-Wesley, 2002.
- [31] P. C. Clements, "Coming Attractions in Software Architecture," Software Engineering Institute - Carnegie Mellon University, Pittsburgh, Technical Report CMU/SEI-96-TR-008, January 1996.
- [32] CMU/SEI. (2003, Dec). Software Technology Roadmap - Glossary. Software Engineering Institute [Online]. Available: <http://www.sei.cmu.edu/str/indexes/glossary/>
- [33] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Method: The Fusion Method*. London: Prentice Hall, 1994.
- [34] D. Consortium. (2003, Feb. 25). DSDM Delivering Agile Business Solutions on Time. Dynamic Systems Development Method Ltd. [Online]. Available: <http://www.dsdm.org/en/default.asp>
- [35] I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau, *Proceedings ICSE 4th international workshop on Component-Based Software Engineering*. Los Alamitos: IEEE Computer Society Press, 2001.
- [36] K. Czarnecki and U. W. Eisenacker, *Generative Programming: Methods, Tools, and Applications*. Reading, Massachusetts: Addison- Wesley, 2000.
- [37] D. Dori, *Object-Process Methodology - A Holistic Systems Paradigm*. Heidelberg, New York: Springer Verlag, 2002.
- [38] D. Dori, "Why Significant UML Change is Unlikely," *Communications of the ACM*, vol. 45, pp. 83-85, November 2002.
- [39] R. G. Dromey, "Cornering the Chimera," *IEEE Software*, vol. 13, pp. 33-43, January 1996.
- [40] D. F. D'Souza and A. C. Wills, *Objects, Components, and Frameworks with UML : The Catalysis Approach*. Reading, Massachusetts: Addison Wesley, 1998.
- [41] A. Egyed, "Reasoning About Trace Dependencies in a Multi-Dimensional Space," in *Proc. 1st International Workshop on Traceability in Emerging Forms of Software Engineering (In conjunction with 17th ASE)*, 2002.

- [42] EIA, "Processes for Engineering a System," Electronic Industries Alliance, Standard ANSI/EIA-632, January 1999.
- [43] J. Estublier and J.-M. Favre, "Component Models and Technology," in *Building Reliable Component-Based Software Systems*, I. Crnkovic and M. Larsson, Eds. London: Artech House Publishers, 2002.
- [44] D. C. Fallside, "XML Schema Part 0: Primer," World Wide Web Consortium (W3C), W3C Recommendation, 2 May 2001.
- [45] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Practical and Rigorous Approach*, 2nd ed. London: Intl. Thomson Computer Press, 1996.
- [46] P. Fettke and P. Loos, "Specification of Business Components," in *Proc. Net.ObjectDays 2002: 3. vereinigte GI Fachtagung "Objektorientierte Programmierung für die vernetzte Welt"*, 2002.
- [47] R. Filman, S. Barrett, D. Lee, and T. Linden, "Inserting Ilities By Controlling Communications," *Communications of the ACM*, vol. 45, pp. 116-122, January 2002.
- [48] C. Frei, T. Kotic, and O. Preiss, "XML Schema - Are We On The Right Track?," *submitted to IEEE Computer*, 2003.
- [49] C. Ghezzi, M. Jazayeri, and D. Mandrioli, *Fundamentals of Software Engineering*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1991.
- [50] N. Goldstein and J. Alger, *Developing Object-Oriented Software for the Macintosh: Analysis, Design, and Programming*. Reading, MA: Addison-Wesley, 1992.
- [51] P. Gutierrez and M. Escalante, "CBSE: State of the Practice," CBSEnet (IST-2001-35485), Technical Report D8.1 CBSE State of the Practice.doc, 05 May 2003.
- [52] D. Hamlet, D. Mason, and D. Voit, "Theory of Software Reliability Based on Components," in *Proc. International Conference on Software Engineering (ICSE)*, 2001, pp. 361-370.
- [53] C. Hartshorne, P. Weiss, and A. Burks, "Collected Papers of Charles Sanders Peirce," 1931-1958 ed. Cambridge, MA: Harvard University Press, 1997.
- [54] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, pp. 576-583, 1969.
- [55] E. Hochmüller, "Towards the Proper Integration of Extra-Functional Requirements," *The Australian Journal of Information Systems*, vol. 7 (Special Edition - Requirements Engineering), 1999.
- [56] D. Howe. (2002, August). Free On-line Dictionary of Computing (FOLDOC). Imperial College Department of Computing [Online]. Available: <http://foldoc.doc.ic.ac.uk/foldoc/index.html>
- [57] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," Institute of Electrical and Electronics Engineers, Inc., International Standard IEEE Std 610.12-1990.
- [58] IEEE, "IEEE Guide for Developing System Requirements Specifications," Institute of Electrical and Electronics Engineers, Inc. IEEE Std 1233-1998.
- [59] IEEE, "IEEE Recommended Practice for Software Requirements Specifications," Institute of Electrical and Electronics Engineers, Inc. IEEE Std 830-1998.
- [60] IEEE, "IEEE Standard for a Software Quality Metrics Methodology," Institute of Electrical and Electronics Engineers, Inc., International Standard IEEE Std 1061-1998.
- [61] IEEE, "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems," Institute of Electrical and Electronics Engineers, Inc. IEEE Std 1471-2000.
- [62] INCOSE. (2003, March 10). What Is Systems Engineering? International Council on Systems Engineering [Online]. Available: <http://www.incose.org/whatis.html>
- [63] ISO/IEC, "Software engineering - Product quality - Part1: Quality model," ISO/IEC, International Standard 9126-1:2001(E).
- [64] ISO/IEC, "Software engineering - Product quality - Part2: External metrics," ISO/IEC, Geneva, Technical Report TR 9126-2:2003(E).

- [65] ISO/IEC, "Software engineering - Product quality - Part3: Internal metrics," ISO/IEC, Technical Report TR 9126-3:2003(E).
- [66] E. E. Jacobsen, B. B. Kristensen, and P. Nowack, "Architecture = Abstractions over Software," in *Proc. 32nd International Conference on Technology of Object-Oriented Languages (TOOLS)*, 1999.
- [67] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse - Architecture, Process and Organization for Business Success*. Reading, Massachusetts: Addison Wesley, 1997.
- [68] G. Kiczales, J. Irwin, J. Lamping, J.-M. Loingtier, C. V. Lopes, C. Maeda, and A. Mendhekar, "Aspect-Oriented Programming," *ACM Computing Survey*, vol. 28, pp. Article 154, 1996.
- [69] M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, and H. Lipson, "Attribute-Based Architecture Styles," in *Proc. First Working IFIP Conference on Software Architecture*, 1999, pp. 225-243.
- [70] G. J. Klir, *Facets of Systems Science*. vol. 15, 2nd ed. New York: Kluwer Academics / Plenum Publishers, 2001.
- [71] P. B. Kruchten, "The 4+1 View of Model Architecture," *IEEE Software*, vol. 12, pp. 42-50, 1995.
- [72] A. Kuntzmann-Combelles and P. Kruchten, "The Rational Unified Process - An Enabler for Higher Process Maturity," Rational Software Corporation, Canada, White Paper, 2001.
- [73] G. Lakoff, *Women, Fire, and Dangerous Things*. Chicago: The University of Chicago Press, 1987.
- [74] F. v. Latum, R. v. Solingen, M. Oivo, B. Hoisl, D. Rombach, and G. Ruhe, "Adopting GQM-Based Measurement in an Industrial Environment," *IEEE Software*, vol. 15, pp. 78-86, 1998.
- [75] F. Lüders, K.-K. Lau, and S.-M. Ho, "Specification of Software Components," in *Building Reliable Component-Based Software Systems*, I. Crnkovic and M. Larsson, Eds. London: Artech House Publishers, 2002, pp. 23-38.
- [76] H. Lykins, S. Friedenthal, and A. Meilich, "Adapting UML for an Object Oriented Systems Engineering Method (OOSEM)," in *Proc. International Council on Systems Engineering 10th Annual Symposium (INCOSE 2000)*, 2000.
- [77] M. Mannion and B. Keepence, "SMART Requirements," *ACM Software Engineering Notes*, vol. 20, pp. 42-47, April 1995.
- [78] F. Manola, "Providing Systemic Properties (Ilities) and Quality of Service in Component-Based Systems," Object Services and Consulting, Inc., Technical Report, February 1999.
- [79] J. Martin-Albo, M. F. Bertoa, C. Calero, A. Vallecillo, A. Cechich, and M. Piattini, "CQM: A Software Component Metric Classification Model," in *Proc. 7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'2003)*, 2003.
- [80] J. A. McCall, P. K. Richards, and G. F. Walters, "Factors in Software Quality," US Rome Air Development Center, Technical Report RADC TR-77-369 Vol.I/II/III/1977.
- [81] B. McLaughlin, *Java & XML*, 2nd ed. Cambridge: O'Reilly & Associates, 2001.
- [82] N. Medvidovic, P. Gruenbacher, A. Egyed, and B. W. Boehm, "Software Model Connectors: Bridging Models across the Software Lifecycle," in *Proc. 13th International Conference on Software Engineering and Knowledge Engineering*, 2001, pp. 387-396.
- [83] R. Medvidovic and R. A. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, vol. 26, pp. 70-93, 2000.
- [84] Merriam-Webster, *Collegiate Dictionary*: Merriam-Webster Online, 2003.
- [85] B. Meyer, *Eiffel: The Language*, 2 ed. London: Prentice Hall, 1992.
- [86] M. Mezini and K. Lieberherr, "Adaptive plug-and-play components for evolutionary software development," in *Proc. Conference on Object Oriented Programming Systems Languages and Applications*, 1998, pp. 97-116.

- [87] G. A. Miller. (2002). WordNet®. Cognitive Science Laboratory, Princeton University [Online]. Available: <http://www.cogsci.princeton.edu/~wn/>
- [88] J. G. Miller, *Living Systems*. Colorado: University Press of Colorado, 1995.
- [89] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan, "Architectural Styles, Design Patterns, and Objects," *IEEE Software*, vol. 14, pp. 43-52, 1997.
- [90] J. Mylopoulos, L. Chung, and E. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis," *Communications of the ACM*, vol. 42, pp. 31-37, 1999.
- [91] P. Nesi. (2002, August). Computer Science Dictionary - Software Engineering Terms. University of Florence [Online]. Available: <http://hpcn.dsi.unifi.it/~dictionary/>
- [92] O. Nierstrasz and D. Tsichritzis, *Object-Oriented Software Composition*. London: Prentice Hall, 1995.
- [93] NIST. (2002, May 9). The NIST Reference of Units, Constants, and Uncertainties. [Online]. Available: <http://physics.nist.gov/cuu/Units/index.html>
- [94] D. W. Oliver, "Concept Model for System Engineering," Object Management Group, Inc., Wakefield, R.I., Presentation of Draft Model, Feb. 27 2003.
- [95] OMG. (2002, June 15). Model Driven Architecture. Object Management Group, Inc. [Online]. Available: <http://www.omg.org/mda/>
- [96] OMG, "OMG-Unified Modeling Language, V1.4," Object Management Group, Inc., Specification, September 2001.
- [97] OMG. (2003, Jan). Systems Engineering Domain Special Interest Group. Object Management Group [Online]. Available: <http://syseng.omg.org/>
- [98] OMG, "Requirements Analysis For UML for Systems Engineering (SE)," Object Management Group, Inc., Needham, MA, Draft V0.4 syseng/2003-02-01, Draft V0.4, Nov. 12 2002.
- [99] OMG, "UML Profile for Enterprise Distributed Object Computing," Object Management Group, Inc., OMG Adopted Specification ptc/02-02-05, February 2002.
- [100] OMG, "UML for Systems Engineering - Request for Proposal," Object Management Group, Inc., Needham, MA, Draft RFP, March 1 2003.
- [101] OMG-XMI-RTF, "XML Metadata Interchange (XMI) Version 1.1," Object Management Group OMG Document ad/99-10-02, October 25 1999.
- [102] D. Parnas, "Information Distribution Aspects of Design Methodology," in *Proc. 1971 IFIP Congress*, 1971, pp. 339-344.
- [103] D. L. Parnas and D. M. Weiss, "Active Design Reviews: Principles and Practices," *Journal of Systems and Software*, vol. 7, pp. 259-265, December 1987.
- [104] D. A. Peled, *Software Reliability Methods*. New York: Springer-Verlag, 2001.
- [105] M. Peleg and D. Dori, "From Object-Process Diagrams to a Natural Object-Process Language," in *Proc. Next Generation Information Technologies and Systems: 5th International Workshop*, 2002, pp. 221-228.
- [106] D. S. Platt, *Understanding COM+*. Redmond, WA: Microsoft Press, 1999.
- [107] K. Pohl and P. Haumer, "Modeling Contextual Information about Scenarios," in *Proc. Third International Workshop on Requirements Engineering: Foundation for Software Quality RESFQ*, 1997.
- [108] O. Preiss and A. Frei, "Systematic Reuse Concept Report," ABB Switzerland Ltd., Corporate Research, Baden-Daettwil, Internal Technical Report ABB CH-RD 2002-18, June 10 2002.
- [109] O. Preiss and M. Naedele, "Architectural Support for Reuse: A Case Study in Industrial Automation," in *Building Reliable Component-Based Software Systems*, I. Crnkovic and M. Larsson, Eds. London: Artech House Publishers, 2002, pp. 325-353.
- [110] O. Preiss, A. Shah, and A. Wegmann, "Generating Synchronization Contracts for Web Services," in *Proc. IRMA International Conference*, 2003, pp. 593 - 596.

- [111] O. Preiss, A. Wegmann, and J. Wong, "On Quality Attribute Based Software Engineering," in *Proc. 27th Euromicro Conference*, 2001, pp. 114-120.
- [112] R. S. Pressman, *Software Engineering: A Practitioner's approach*, 5 ed. Boston: McGraw Hill, 2001.
- [113] S. Puroo and V. Vaishnavi, "Product Metrics for Object-Oriented Systems," *ACM Computing Surveys*, vol. 35, pp. 191-221, June 2003.
- [114] QCCS. (2002, Jan). Quality Controlled Component-Based Software Development. European Community IST Project-1999-20122 [Online]. Available: <http://www.qccs.org/>
- [115] B. Ramesh, "Factors Influencing Requirements Traceability Practice," *Communications of the ACM*, vol. 41, pp. 37 - 44, 1998.
- [116] Rational, "RAS - Reusable Asset Specification," Rational Software Corporation, Web document RAS 2001.09.04, 2001.
- [117] W. N. Robinson and S. Volkov, "A Meta-Model for Restructuring Stakeholder Requirements," in *Proc. International Conference on Software Engineering (ICSE 97)*, 1997.
- [118] R. Rosen, "Some Comments on systems and system theory," *International Journal of General Systems*, vol. 13, pp. 1-3, 1986.
- [119] J. R. Rumbaugh, M. R. Blaha, W. Lorensen, F. Eddy, and W. Premerlani, *Object Oriented Modeling and Design*, 1st ed. London: Prentice Hall, 1991.
- [120] B. Russell. (2002, July 20). The Problems of Philosophy. Home University Library [Online]. Available: <http://www.ditext.com/russell/russell.html>
- [121] M. A. Sasse, Eliciting and Describing Users' Models of Computer Systems, PhD, University of Birmingham, School of Computer Science, Birmingham, 1997
- [122] P. C. Scott and S. Rose, "Integrated development for computer-based systems," in *Proc. International Conference and Workshop on Engineering of Computer-Based Systems*, 1997, pp. 414 -420.
- [123] SEI. (2003, Nov). Capability Maturity Model® for Software (SW-CMM®). Carnegie Mellon Software Engineering Institute [Online]. Available: <http://www.sei.cmu.edu/cmm/>
- [124] S. Sendall and A. Strohmeier, "Specifying Concurrent System Behavior and Timing Constraints Using OCL and UML," in *Proc. UML 2001 - The Unified Modeling Language: Modeling Languages, Concepts and Tools*, 2001, pp. 391-405.
- [125] M. Shaw, "Truth vs Knowledge: The Difference Between What a Component Does and What We Know It Does," in *Proc. 8th International Workshop on Software Specification and Design (IWSSD-8)*, 1996, pp. 181-185.
- [126] H. A. Simon, *The Sciences of the Artificial*, Third ed. Cambridge, Massachusetts: The MIT Press, 1999.
- [127] C. Sluman, J. Tucker, J. P. LeBlanc, and B. Wood, "Quality of Service (QoS) OMG Green Paper," Object Management Group, OMG Green Paper Version 0.4a, June 12 1997.
- [128] I. Sommerville, *Software Engineering*, 6th ed. Harlow: Addison-Wesley, 2001.
- [129] I. Sommerville and P. Sawyer, "Viewpoints: principles, problems and a practical approach to requirements engineering," *Annals of Software Engineering*, vol. March, 1996.
- [130] C. Swoyer. (2002, July). Properties. The Metaphysics Research Lab, Stanford University [Online]. Available: <http://www.science.uva.nl/~seop/entries/properties/>
- [131] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*. Reading, Massachusetts: Addison-Wesley, 1998.
- [132] A. Tarski, *Logic, Semantics, Metamathematics*: Oxford University Press, 1956.
- [133] U2-Partners, "Unified Modeling Language: Superstructure, Version 2.0," Submission to RFP 2nd revised submission to OMG RFP ad/00-09-02, Jan 6 2003.
- [134] P. Valéry, *Cahiers*. vol. Tome 1. Paris: Gallimard (collection "La Pléiade"), 1975.

- [135] A. Villemeur, *Reliability, Availability, Maintainability And Safety Assessment*. vol. 1. Chichester, UK: John Wiley & Sons, 1991.
- [136] K. Wallnau, S. Hissum, and R. Seacord, *Building Systems from Commercial Components*. Boston: Addison-Wesley, 2002.
- [137] W.-L. Wang, "Beware the Engineering Metaphor," *Communications of the ACM*, vol. 45, pp. 27-29, May 2002.
- [138] D. Watkins, "Using Interface Definition Languages to Support Path Expressions and Programming by Contract," in *Proc. Technology of Object-Oriented Languages and Systems (TOOLS 26)*, 1998, pp. 308-319.
- [139] S. Watman, "Requirements Specification for Reusable Asset Viewer," ABB Switzerland - Corporate Research, Baden-Daettwil, Internal Technical Report CRID20016-02-01, October 22 2002.
- [140] A. Wegmann, "On the Systemic Enterprise Architecture Methodology (SEAM)," in *Proc. 5th International Conference on Enterprise Information Systems (ICEIS)*, 2003, pp. 483-490.
- [141] A. Wegmann and O. Preiss, "Strengthening MDA by Drawing from the Living System Theory," in *Proc. <<UML'2002>> Workshop in Software Model Engineering (WiSME@UML2002)*, 2002.
- [142] A. Wegmann and O. Preiss, "MDA in Enterprise Architecture? The Living System Theory to the Rescue...," in *Proc. 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003)*, 2003, pp. 2-13.
- [143] S. White, M. Alford, J. Holtzman, S. Kuehl, B. McCay, D. Oliver, D. Owens, C. Tully, and A. Willey, "Systems Engineering of Computer-Based Systems," *IEEE Computer*, vol. 26, pp. 54-65, 1993.
- [144] M. Wiedmann, H. Buchwald, and D. Seese, "Design by Contract in Java - a Roadmap to Excellence in Trusted Components," *INFORMATIK*, pp. 9-14, 2000.
- [145] E. N. Zalta, "Stanford Encyclopedia of Philosophy," vol. 2002, Summer 2002 Edition ed. Stanford, CA: The Metaphysics Research Lab, Stanford University, 2002.

Appendix A: Glossary of Terms and Abbreviations

Appendix A provides three tables:

- A glossary of general terms and what their assumed definition is in the context of this thesis report (Table 23)
- A list of abbreviations (Table 24)
- A table with informal definitions for many high-level non-functional properties (Table 25).

Table 23: Glossary of General Terms provides a list of terms, which either are less frequently used, especially in the systems- and software engineering context, or are overused and therefore need to have a defined meaning in this thesis report.

Table 23: Glossary of General Terms

Conceptual System	A set of interrelated units, where the units and their interrelations are concepts, i.e. entities that are conceived in mind. See also System.
Concrete System	A set of interrelated units, where the set consists of a non-random accumulation of matter and energy. See also System.
Context	Used in a linguistic sense: Discourse that surrounds a language unit and helps to determine its interpretation [87], “the parts of a discourse that surround a word or passage and can throw light on its meaning” [84] Used in a circumstance, physical environment sense: the interrelated conditions in which something exists or occurs [84]
Designed System	A conceptual <i>or</i> concrete system made by man.
Entity	Something that has conceptual or concrete reality. [84]
Epistemology	The study or theory of the nature and grounds of knowledge. Note, while ontology tries to explain the nature of the world, epistemology tries to explain the nature of our experience of the world.
Etymology	The study of words (their history and development)
Isomorphism vs. homomorphism	Acc. to [84]: Isomorphism is therefore (a) the quality or state of being isomorphic ⁹⁾ , or (b) a one-to-one correspondence between two mathematical sets; <i>especially</i> : a homomorphism that is one-to-one. Homomorphism: A mapping of a mathematical set onto another set or itself in such a way that the result obtained by applying the operations to elements of the first set is mapped onto the result obtained by applying the corresponding operations to their

	<p>respective images in the second set.</p> <p>^{*)} Isomorphic: being of identical or similar form, shape, or structure</p>
Ontology	<p>A theory about the nature of being and existence. The term is borrowed in computer science and especially in AI to stand for a specification of a representational vocabulary for a shared domain of discourse. In that context, what "exists" is that which can be represented. When the knowledge of a domain is represented in a declarative formalism, the set of objects that can be represented is called the universe of discourse. This set of objects, and the describable relationships among them, are reflected in the representational vocabulary.</p>
Semantics	<p>The study of language meaning</p>
Software Engineering	<p>The theory and practice of engineering principles to construct and maintain software.</p>
Stakeholder	<p>A stakeholder is a role that represents groups of people who are interested in the same phenomenon. These people are said to have stakes on the phenomenon.</p>
Subsystem	<p>A conceptual system that captures/models behavior that is jointly provided by collaborating concrete components. In the LST sense, an abstraction that carries out a particular process for a living system.</p>
System	<p>For short: A set of interrelated units that can be considered as a whole.</p> <p>More elaborate: "A system is a set of interacting units with relationships among them. The word "set" implies that the units have some common properties. These common properties are essential if the units are to interact or have relationships. The state of each unit is constrained by, conditioned by, or dependent on the state of other units. The units are coupled. Moreover, there is at least one measure of the sum of its units which is larger than the sum of that measure of its units." [88]</p>
Systems Engineering	<p>The theory and application of engineering principles to construct and maintain designed systems.</p>
Teleology	<p>Acc. to [84]:</p> <p>1 a : the study of evidences of design in nature b : a doctrine (as in vitalism) that ends are immanent in nature c : a doctrine explaining phenomena by final causes</p> <p>2 : the fact or character attributed to nature or natural processes of being directed toward an end or shaped by a purpose</p> <p>3 : the use of design or purpose as an explanation of natural phenomena</p>
Thing	<p>A linguistic placeholder for something that may be named and only defined in context.</p>
Universe of Discourse	<p>An inclusive set of things that is tacitly implied or explicitly delineated as the subject of a statement, discourse, or theory.</p>

	Practically, this is everything stated or assumed in a discussion.
View	A concept to partition model representations into viewpoint related sets.
Viewpoint	The mental position from which things are viewed [87]

Table 24: List of Abbreviations

ABB	Asea Brown Boveri
ABD	Architecture-Based Design [9]
AIP	Aspect Integrator Platform (ABB)
aka	also known as
ASME	American Society of Mechanical Engineers
ATM	Automatic Teller Machine
CASE	Computer-Aided Software Engineering
CBD	Component-Based Development
CBSE	Component-Based Software Engineering
COM	Component Object Model (Microsoft)
CORBA	Common Object Request Broker Architecture (OMG)
EDI	Electronic Data Interchange
EJB	Enterprise Java Beans
GUI	Graphical User Interface
IDL	Interface Definition Language
IT	Information Technology
LOC	Lines Of Code
LST	Living Systems Theory (Miller [88])
MOF	Meta-Object Facility
OCL	Object Constraint Language (part of the UML)
OLE	Object Linking and Embedding (Microsoft)
OO	Object Oriented
OOP	Object Oriented Programming
OPC	OLE for Process Control
PC	Personal Computer
QoS	Quality of Service
RAS	Reusable Asset Specification
UML	Unified Modeling Language

UoD	Universe of Discourse
XML	eXtended Markup Language

Table 25 provides a list of alphabetically ordered, referenced definitions of mainly non-functional properties as a convenience for the reader of this thesis report.

Table 25: Informal Definition of High-Level, Non-Functional Properties

Property Name	Informal Definition
Accessibility	The ability of software applications to be accessible to users with impairments. Related to Denial of Service, degree to which the software system protects system functions or service from being denied to the user the. [32] Related to Reusability, the degree to which a software system or component facilitates the selective use of its components. [32]
Accuracy	The capability of the software product to provide the right or agreed results or effects with the needed degree of precision. [63]
Adaptability	The capability of the software product to be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered.
Administrability	The ease with which deployed software can be administrated by system administrators, especially in a distributed multi-client environment.
Analyzability	The capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified. [63]
Attractiveness	The capability of the software product to be attractive to the user. [63]
Availability	The degree to which a system suffers degradation or interruption in its service to the customer as a consequence of failures of one or more of its parts [56]. The ability of a system to be up and running. Usually measured by the length of time between failures as well as by how quickly the system can resume operation after a failure. [14]
Changability	The capability of the software product to enable a specified modification to be implemented. [63] See also modifiability.
Co-existence	The capability of the software product to co-exist with other independent software in a common environment sharing common resources. [63]
Coherency	To ability of a system or of particular parts of it to appear or be used coherent (as one “thing”, in one similar “way”) despite underlying differences. Related to computer data, coherency refers to the ability of two or more physical storage locations to accurately reflect the exact same value for a single (logical) data item.
Composeability	See Integrability
Degradeability	The ability of the software product to still provide its specified, or a well-defined reduced form, of its capabilities in the presence of a degrading environment and thus in the presence of fewer resources.
Dependability	The capabilities of a system which in its sum allow a user to depend on it. The

	collective term used to describe the availability and its influencing factors such as reliability, etc. (IEC TC65).
Deployability	The capability to deployment, migration. The ease with which a software product can be deployed into its execution environment, or more generally, the ease of migration of software to a geographically dispersed environment.
Distributability	The ability of a software product to be easily decomposed so that parts of the software can run on physically distributed but interconnected computing resources.
Durability	The ability to exist for a long time without significant deterioration. [84] See also Degradability.
Effectiveness	The capability of the software product to enable users to achieve specified goals with accuracy and completeness in a specified context of use. [63]
Efficiency	The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions. [63]
Evolvability	The ease with which a system or component can be modified to take advantage of new software or hardware technologies [32]
Extendibility	The ease with which a system or component can be modified to increase its storage or functional capacity [57]
Extensibility	The ability of a software product to be extended. In the era of software components, <i>dynamic</i> extensibility refers to the ability of a software component or software component based system to have components extended or systems replaced or extended with components at system runtime.
Expandability	See Extendibility [57].
Fault tolerance	The capability of the software product to maintain a specified level of performance in cases of software faults or of infringement of its specified interface. [63]
Footprint	The floor or desk area taken up by a piece of hardware. The amount of disk memory or RAM taken up by a program or file. [56]
Functionality	The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions. [63]
Installability	The capability of the software product to be installed in a specified environment. [63]
Integrability	The ability to make the separately developed components of the system work correctly together [14].
Integrity	The degree to which a system or component prevents unauthorized access to, or modification of, computer programs or data [57]
Interoperability	The capability of the software product to interact with one or more specified systems.[63]
Learnability	The capability of the software product to enable the user to learn its application. [63]
Maintainability	The capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. [63]

Maturity	The capability of the software product to avoid failure as a result of faults in the software. [63]
Memorability	In the context of usability, the ability of the system to facilitate the user in remembering how to do system operations between uses of the system [14]
Mobility	The ability of software to change location in order to continue execution (key words: self-organizing systems, mobile agents). AISB'01 Symposium on Software mobility: http://www.ecs.soton.ac.uk/~lavm/aisb/cfp.html
Modifiability	The ability to make changes quickly and cost effectively [14]
Nomadcity	The support of a system by specific (transparent, integrated and convenient) computing and communication capabilities and services to nomads, i.e. users (or other systems), as they move from place to place. (NOMADIC'97: http://www.tticom.com/nomadic/about.htm)
Openness	The degree to which a system or component complies with standards [32]. The degree with which the software product is open and not proprietary or closed, in the sense that it can be programmatically interfaced by other software through defined interfaces, preferably adhering to some standards. Relates to integrability.
Operability	The capability of the software product to enable the user to operate and control it. [63]
Performance	Refers to the responsiveness of the system, the time required to respond to stimuli /events) or the number of events processed in some time interval [14]
Portability	The capability of the software product to be transferred from one environment to another. [63]
Productivity	The capability of the software product to enable users to expend appropriate amounts of resources in relation to the effectiveness achieved in a specified context of use. [63]
Recoverability	The capability of the software product to re-establish a specified level of performance and recover the data directly affected in the case of a failure. [63]
Reliability	The capability of the software product to maintain a specified level of performance when used under specified conditions. [63]
Replaceability	The capability of the software product to be used in place of another specified software product for the same purpose in the same environment. [63]
Resource Utilization	The capability of the software product to use appropriate amounts and types of resources when the software performs its function under stated conditions. [63]
Reuseability	The capability and provisions of a software artifact to be reused in any other than its originally developed software product.
Safety	The capability of the software product to achieve acceptable levels of risk of harm to people, business, software, property or the environment in a specified context of use. [63]
Satisfaction	The capability of the software product to satisfy users in a specified context of use. [63]
Scaleability	The capability of the software product

Schedulability	The capability of the software product to allow its program execution to be controlled in a defined manner by a scheduler.
Security	The capability of the software product to protect information and data so that unauthorised persons or systems cannot read or modify them and authorised persons or systems are not denied access to them.[63]
Serviceability	The ease with which corrective maintenance or preventative maintenance can be performed on a system (e.g. by a hardware service technician). Higher serviceability improves availability and reduces service cost. [56]
Stability	The capability of the software product to avoid unexpected effects from modifications of the software. [63] A measure for the amount of changes of a software product between two releases [112].
Suitability	The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives. [63]
Survivability	Survivability is the ability of a network computing system to provide essential services in the presence of attacks and failures, and recover full services in a timely manner.
Tailorability	The ability of a software product to provide means to tailor its capabilities according to a given context of use.
Testability	The capability of the software product to enable modified software to be validated. [63] But also, the ease with which software can be made to demonstrate its faults /typically through execution-based) testing. [14]
Timeliness (time behavior)	The capability of the software product to provide appropriate response and processing times and throughput rates when performing its function, under stated conditions. [63]
Understandability	The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use. [63]
Upgradability	The ability of the software product to be substantially enhanced by a newer version with further functionality or otherwise improved capabilities, without a negative impact on its already existing use.
Usability	The capability of the software product to be understood, learned, used, and be attractive to the user, when used under specified conditions.

Appendix B: XML Schema Alternatives for the RAS

For convenience reasons we restate the requirements that we formulated in Section 7.2.3 for a physical representation of the RAS:

R1: It must be amenable to tool processing, because we want to automate the handling of software assets, along the entire reuse process, by a tool infrastructure.

R2: The tool infrastructure shall be independent of the type of software assets processed. I.e. new types of software assets need to be introduced without any change in the tool landscape.

R3: Producers of reusable assets shall be forced to include all required information in an asset description, but they shall be allowed to include more, if they want, in a compliant way. Thus, respective rules and constraints shall be automatically checkable and enforceable with a validation step.

R4: The development of proprietary validation software shall be avoided as far as possible, i.e. standard technology shall be used, where parsers, validation software etc. are readily available.

R5: Software asset descriptions shall remain human read- and interpretable.

R6: All constraints and rules (as discussed under item R3 above) have to be expressed in the RAS profile XML schema, which then enables automatic validation and the usage of commercially available XML validation software.

R7: Relationships to the RAS core metamodel concepts need to be expressed in the RAS profile XML schemas and RAS XML documents for the below reasons:

- We need to be able to validate the introduction of new types of software assets, in the form of new RAS profile XML schemas, against the RAS core metamodel without modification of the existing tool infrastructure (derived from R2).
- We need to be able to accept (i.e. to validate) elements and attributes that are provided by the producer but are exceeding the required minimal information (derived from R3),
- We want humans to be able to interpret RAS XML documents (derived from R5). For example, they shall be able to understand that when they find “development-environment” in the RAS XML document this is meant to be a specific “classification descriptor” (a core metamodel concept) of the asset at hand.

The proposed process of the RAS for creating a new RAS profile XML schema is defined as follows:

- a) Create the RAS profile in form of a Microsoft Word[®] document according to the provided template. The document contains definitions, descriptions, constraints and guidelines on how to package information for an individual software asset description. Note that constraints and rules are captured informally as free text.
- b) Determine what will be extended, the RAS core metamodel or an existing RAS profile.
- c) Use the RAS core XML schema or an appropriate RAS profile XML schema as a starting point and “extend” by editing the schema with the semantics of the new profile.

This approach does not deliver results well suited for tool automation because constraints and rules are informally captured in the RAS profile Word[®] document and can therefore not automatically be enforced (see requirement R3 above). Moreover, the concrete steps of the task of extending the

RAS core XML schema, or an existing RAS profile XML schema, are not specified. Consequently, there are many alternative solutions to create RAS profile XML schemas.

Another problem is the mapping of the three-layered modeling approach of the RAS onto XML (recall Figure 7-3 on page 146). XML essentially is limited to a two-layered approach because it lacks features for building generic type hierarchies. Consequently, automated validation with standard software is only possible between an XML instance file and its single type level XML schema. This means that it is not possible to automatically validate a RAS profile XML schema against the RAS core XML schema with standard software. This is a serious obstacle for requirement R2, which demands that tools shall be implemented in such a way that they can handle all possible types of software assets – even types that are not known at the time of the tool implementation. This requirement implicitly assumes that the “business logic” of a tool shall not depend on specific RAS profiles. Rather, the tool’s logic needs to rely on the generic rules expressed in the RAS core metamodel. From this we infer that for tool automation to be flexible, metamodel information needs to be available, either directly or indirectly, from the RAS XML documents. This confirms that requirement R7 – the availability of metamodel information – is crucial to support the automation of the reuse process with a generic tool infrastructure.

B 1 The Running Example

The following example will serve both as an illustration of the aforementioned problems with the RAS recommendation to create profiles and as an illustration of the different solution alternatives discussed, starting with B 2. In order to keep the example small and because the issues are the same for all four parts of the RAS logical model, we limit our discussion to the *classification* part and the concept of *descriptors*. We picked the classification also, because our quality attribute extension of the RAS is mainly dependent on this part of the RAS.

In Figure B- 1 we illustrate the essence of Figure 7-5 (page 150) with respect to this discussion. It depicts the heart of the classification part of the RAS core metamodel. Paraphrased, it says that an *Asset* has exactly one *Classification* part, consisting of at least one *Descriptor Group* and each descriptor group consists of at least one *Descriptor Name-Value Pair*, which we call *descriptor* throughout the following discussion.



Figure B- 1: RAS Core Metamodel of the Classification Part

Examples of descriptor names could be *keyword* (for all kinds of software assets), *runtime environment* (for executable software assets), *programming language* (for all software assets, where the source code is relevant), etc. One concrete instance of such a name/value pair descriptor could be [“programming language”, “C++”]. Descriptor groups as shown in Figure B- 1 are used to group semantically related descriptors. In order to simplify the following discussions we limit ourselves to one descriptor group called “generic”.

The running example is concerned with a RAS description of software components (such as COM components, JavaBeans, etc.). Consequently, we would like to create a new RAS profile and its related RAS profile XML schema as a blueprint to describe individual software components. An asset of this new type “software component” may contain many artifacts, such as overview document(s), source code file(s), executable file(s), IDL files, etc. In order to enable an interested consumer to search for such an asset and evaluate it, we define several descriptors. Among them are the following three:

Programming Language, Development Environment, and Runtime Environment. They are all conceptual properties, which are ascribed to the software component.

The XML Schema, which corresponds to the RAS core metamodel of the classification part as shown in Figure B- 1, is depicted in Figure B- 2. This figure and all subsequent screen shots are taken from a tool called XMLSpy [3].

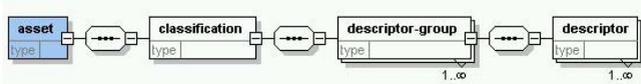


Figure B- 2: RAS Core XML Schema of the Classification Part

The presented RAS core XML schema of the classification part represents, in the form of XML Schema constructs (XML *Elements* depicted as boxes, XML *Sequences* depicted as icons with three dots), the same concepts and relationships with their cardinalities, as does the UML representation shown in Figure B- 1. Neglecting minor differences, the XML Schema shown in Figure B- 2 is in that form part of the downloadable RAS core XML schema.

In order to create the new RAS profile XML schema for the software component asset type, we replace selected metamodel concepts with concrete, asset type-specific terms. Figure B- 3 presents the classification part of such a concrete model for the software component asset type. As can be seen in Figure B- 3, the abstract concepts such as *descriptor group* and *descriptor* are directly replaced by concrete element names like *generic* and *programming-language*, respectively. The name/value pair of a descriptor is represented as follows: The name is represented by the XML Element name, e.g., *programming-language* and the value is represented as an XML Attribute. As an example, see the *Value* attribute in the lower pane of Figure B- 3 for the highlighted *programming-language* element. A compliant RAS XML document of the proposed RAS profile XML schema is shown in Figure B- 4.

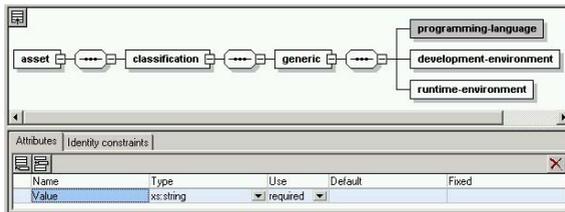


Figure B- 3: RAS Profile XML Schema Based on the Running Example

```

<asset>
  <classification>
    <generic>
      <programming-language value="C++"/>
      <development-environment value="Visual Studio"/>
      <runtime-environment value="COM"/>
    </generic>
  </classification>
</asset>

```

Figure B- 4: Example RAS XML Document Based on the Running Example

Important to note is that the RAS profile XML schema and therefore also the XML document preserve the structural arrangement of the RAS core metamodel but they do not allow explicit identification of XML Schema elements as instances of meta-types. Concretely speaking, it is nowhere declared that *generic* is a RAS descriptor-group and *programming-language* is a RAS descriptor. This sort of information can only be implicitly derived from the structure of the XML document. This means that the parsing software of the file needs to know that a classification contains descriptor-groups and descriptor-groups contain one or more descriptors. Therefore, this knowledge of the structure needs to be built into the parsing software as business logic and is not contained in the RAS profile XML schema or RAS XML document.

Obviously, this approach does not fulfill all of our previously listed requirements. Most importantly, requirement R2 and R7 (asset type independent tool infrastructure and RAS metamodel information) are not fulfilled, but also R5 (human interpretability) is not fulfilled since a human reader cannot easily derive the meta-type of an element. However, all other requirements are covered with this solution: Requirements R3 and R6 are fulfilled because constraints and rules can be expressed in the RAS profile XML schema. For instance the *programming-language* element must be present, its *value* attribute is marked as *required* and needs to be of type *string*. Validation and enforcement of the constraints and rules with commercially available software is possible because software for validating XML documents against an XML Schema is readily available (requirement R4).

Because the RAS does not specify how to define RAS profile XML schemas and because requirement R2 and R7 are too important, we investigated other feasible approaches. In the subsequent Sections we present four possible alternatives with their pros and cons related to our requirements. They are gradual improvements over the variant discussed in this Section. Alternatives one to three try to improve the previously given example in such a way that requirements R2, R5 and R7 are also fulfilled. Therefore, these alternatives include metamodel information in the RAS profile XML schema and thus also in the RAS XML documents. In these three alternatives, metamodel information is expressed through different XML constructs. Alternative one uses XML element names (Section B 2), alternative two uses XML attributes (Section B 3), and alternative three uses XML types (Section B 4). Alternative four (Section B 5) is an attempt towards creating type hierarchies by means of XML complex types. As such, it is an extension of alternative three, but cannot be realized satisfactorily with current XML technology.

B 2 Metamodel Information as XML Element Names

The first alternative solution proposes RAS profile XML schemas where parts of the XML element names contain the metamodel information. The elements of the RAS profile XML schema for this solution are named by the concatenation of “metamodel concept name” and a running number N, as shown in Figure B- 5 (e.g. descriptor-group1, descriptor1, descriptor2, etc.). Note, that the

numbering is required since there cannot be more than one entity with the same name on the same hierarchy level in an XML Schema.

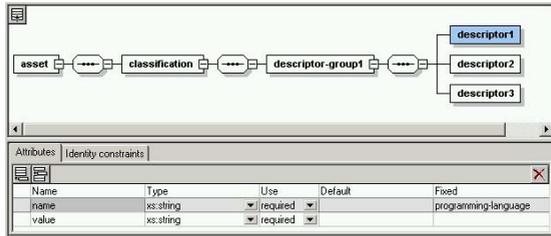


Figure B- 5: RAS Profile XML Schema Based on Alternative One

```

<asset>
  <classification>
    <descriptor-group1 name="generic">
      <descriptor1 name="programming-language"
        value="C++"/>
      <descriptor2 name="development-environment"
        value="Visual Studio"/>
      <descriptor3 name="runtime-environment"
        value="COM"/>
    </descriptor-group1>
  </classification>
</asset>

```

Figure B- 6: Example RAS XML Document of Alternative One

The corresponding RAS XML document of this alternative is shown in Figure B- 6. All the numbered elements have a *name* attribute (e.g., *name = generic* for the descriptor-group1 or *name = programming-language* for the descriptor1). The advantage is that metamodel information is explicitly available in the RAS XML document and can be retrieved, albeit awkwardly, by parsing software.

This alternative fulfills basically all our requirements with the exception of the profile validation against the core metamodel by standard validation software. However, the disadvantage is that the numbering is cumbersome and the human readability of the instance file is lower in quality than in the example presented in Section B 1. Not to mention that the numbers need to be stripped in the parsing software to obtain the metamodel concepts.

B 3 Metamodel Information as XML Attributes

Instead of putting the metamodel information into the element names of the RAS profile XML schema, the second alternative solution proposes to keep meaningful element names, as in the example presented at the beginning of Section B 1. In addition, we introduce a mandatory attribute for each element. This attribute shall contain a reference to the metamodel concept. Figure B- 7 shows, besides *value*, a second attribute for the selected *programming-language* element called *ras-type*. The latter shall make an explicit and fixed reference to the metamodel concept, in this case to *descriptor*.

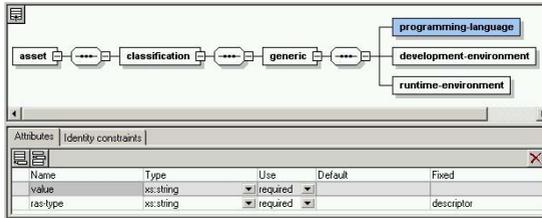


Figure B- 7: RAS Profile XML Schema Based on Alternative Two

Figure B- 8 depicts the RAS XML document derived from such a RAS profile XML schema. Not surprisingly, it holds all the references to the metamodel concepts.

```

<asset ras-type="asset">
  <classification ras-type="classification">
    <generic ras-type="descriptor-group">
      <programming-language value="C++"
        ras-type="descriptor"/>
      <development-environment value="Visual Studio"
        ras-type="descriptor"/>
      <runtime-environment value="COM"
        ras-type="descriptor"/>
    </generic>
  </classification>
</asset>

```

Figure B- 8: Example RAS XML Document of Alternative Two

This alternative fulfills the same requirements as the previous one. The specific advantage of this solution, compared with the first alternative presented in Section B 2, is that all metamodel concepts are always directly available as an attribute of the corresponding element. No special treatment, e.g. number stripping, is required. A minor disadvantage is that XML documents are more verbose and thus bigger than those of the previous alternative.

Alternative two - XML attribute based metamodel references - is the one we recommend and also our choice for the company-specific profiles mentioned in Section 7.3 above.

B 4 Metamodel Information as Types

This alternative solution keeps a reference to the metamodel information in the form of XML types. XML supports simple and complex types. Each element can be assigned a type in an XML schema. Therefore, we can create independent *global* types that represent the metamodel concepts and assign these types to the corresponding elements in the RAS profile XML schema.

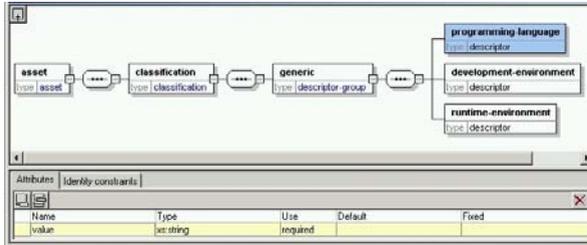


Figure B- 9: RAS Profile XML Schema Based on Alternative Three

The screen shot, shown in Figure B- 9, depicts the type information in an additional rectangle in each of the element boxes. In contrast to the previous two alternatives with element names and attributes as metamodel references, the type information is only available in the XML Schema. We cannot see the type information in the RAS XML document (Figure B- 10).

```

<asset>
  <classification>
    <generic>
      <programming-language value="C++"/>
      <development-environment value="Visual Studio"/>
      <runtime-environment value="COM"/>
    </generic>
  </classification>
</asset>

```

Figure B- 10: Example RAS XML Document of Alternative Three

Actually, Figure B- 10 is identical to Figure B- 4. The latter depicted the RAS XML instance file of the problem example in Section B 1. But opposed to that example, this alternative has the advantage that the metamodel information can be retrieved programmatically from the RAS profile XML schema. Unfortunately, at the time of investigation there seems to be no off-the-shelf XML parsing software available that provides features to obtain type related information about XML elements ([81], p.448-449). However, the metamodel information can be retrieved from the RAS profile XML schema with proprietary implemented functionality because XML schemas are also only XML documents. Thus, any XML parser software can be used to retrieve data from an XML schema. And, albeit quite cumbersome, it is possible to read an element from a RAS XML document and find the appropriate entry in the RAS profile XML schema to get the type information and thus the wanted metamodel information. It is expected that an integrated functionality that retrieves the type information of an element directly will be available with future XML parser software.

One obvious disadvantage of this alternative is that human interpretability is strongly reduced. Metamodel information cannot be directly seen in RAS XML documents. Rather, two files need to be viewed concurrently to get the metamodel information. Therefore, this alternative does not fully fulfill requirement R5. R5 is insofar important to us as that we expect asset producers to frequently edit XML documents manually, i.e. by means of more or less standard text editors. Missing metamodel references increase the risk that non-compliant files are edited.

B 5 Complex Type Hierarchies

All previously presented alternatives have in common that metamodel information is explicitly or implicitly expressed through references to core metamodel concepts by means of XML constructs. However, the requirement for an automated validation of RAS profile XML schemas against the RAS core metamodel is not solved completely. This would be needed to be able to create new profiles without changing the tool infrastructure. For the previously presented alternatives it means, that either every RAS profile XML schema is simply assumed to be compliant to the RAS core metamodel or a proprietary validation step of the RAS profile XML schema is required. What we would like is to validate one XML schema (that of a profile) against another one (that of the RAS core metamodel). In other words, we need formal relationships for what we annotated with a “?” in Figure 7-3 (page 146). One way of realizing such a formal relationship is to define abstract type hierarchies. XML elements should be able to instantiate any type in such a hierarchy. Consequently, an instance would inherit all the constraints on such a type.

This idea of a type hierarchy-based solution is a natural extension of the previous alternative in that the whole RAS core metamodel is built as one complex XML type (instead of independent global types as in the previous alternative). For instance, a complex type *asset* could be defined, which consists of four elements named *overview*, *classification*, *solution*, and *usage*. They, in turn, would be complex types. The *classification* element would be of a complex type containing 1 to n elements of type *descriptor-group* and so on.

This approach sounds promising and would clearly be the most elegant solution, but is not realizable because of the current limitations of the XML. Firstly, XML is limited in expressing constraints between types, such as for instance: “an element of type *descriptor-group* may only contain elements of type *descriptor*”. Secondly, the components of a complex type must be XML elements and cannot be abstract “anonymous” types again. Nesting of complex types is not possible. The name of a component is therefore fixed and cannot be changed by an instance of such a complex type. For example, if we have a complex type *classification* defined as containing a component *descriptor-group*, the latter must be an XML element. An XML document would then contain an XML element *classification* with a named component *descriptor-group*. But it would not be possible to call this component “generic”, as we would like to call it given our example.

Because standardized data exchange of non-trivial data structures will be increasingly important, we have summarized the problems with complex types and several more shortcomings of the current XML Schema standard in [48].

Appendix C: Complete Meta-Model of System and Properties

Figure C-1 depicts the important concepts and concept relationships of Figure 4-4, Figure 4-7, Figure 4-8, Figure 4-12, and Figure 4-15 in one representation. This representation therefore captures our meta-model of (software-intensive) systems and properties, which was discussed in separate pieces in Part II.

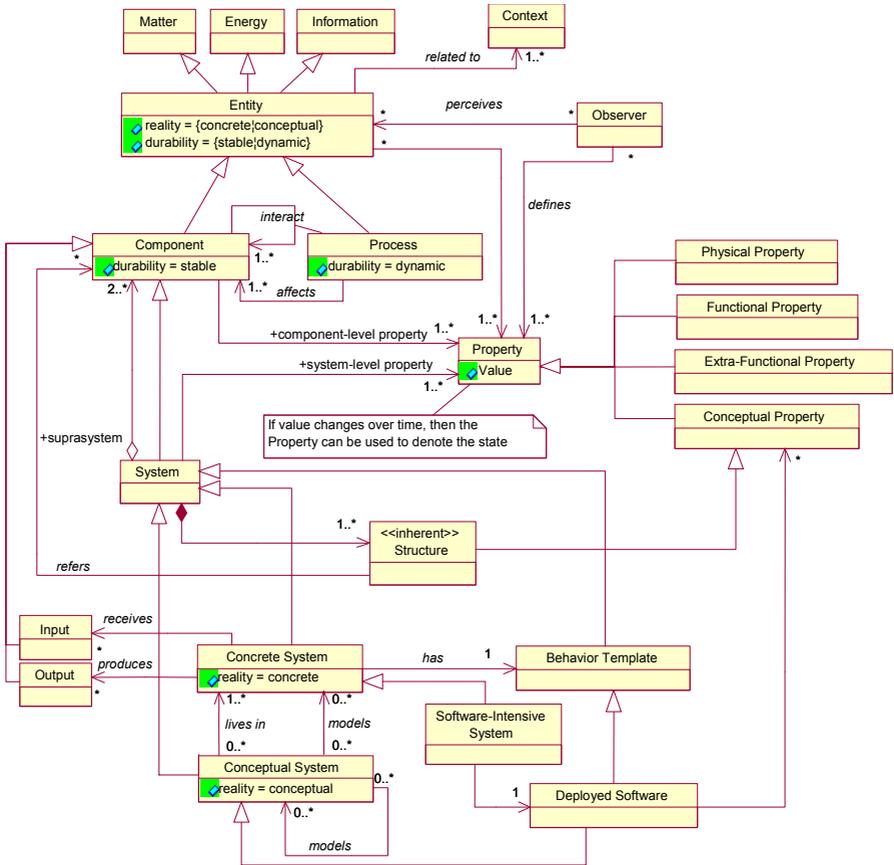


Figure C-1: Complete Meta-Model of System and Properties