Categorizations of Product-related Requirements in Practice

Observations and Improvements

Jonas Eckhardt

Technische
Universität
München

Institut für Informatik
der Technischen Universität München

Categorizations of Product-related Requirements in Practice

Observations and Improvements

Jonas Eckhardt

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Prof. Dr. Florian Matthes
Prüfer der Dissertation:	
1.	Prof. Dr. Dr. h.c. Manfred Broy
2.	Prof. Dr. Klaus Pohl, Universität Duisburg-Essen

Die Dissertation wurde am 18.05.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 13.10.2017 angenommen.

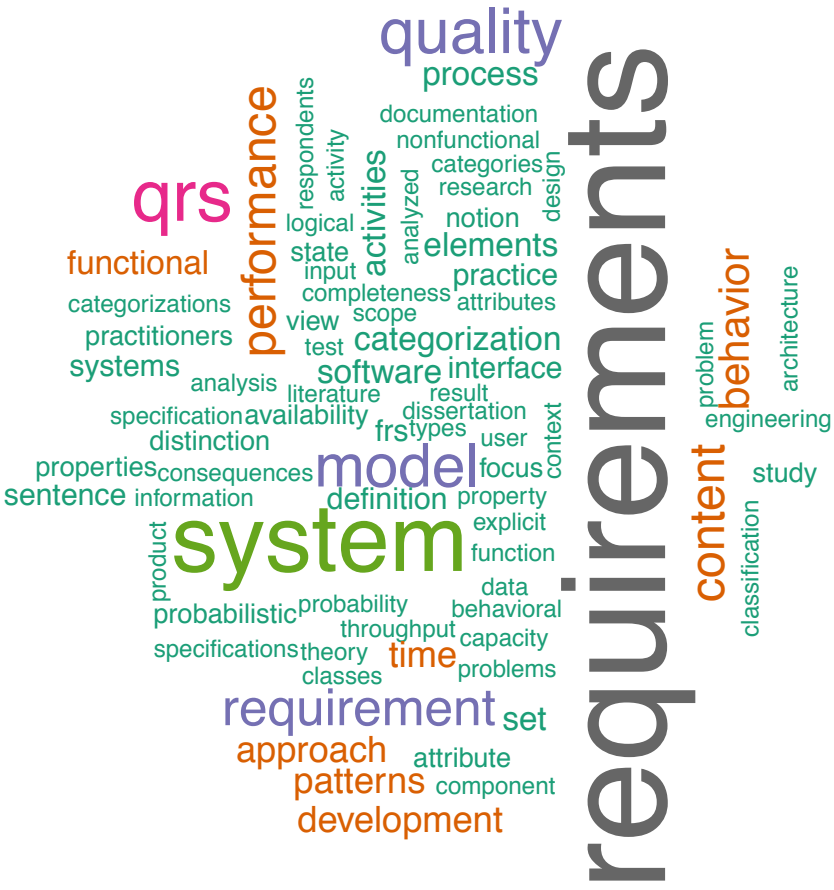
Abstract

There is a significant discussion on categorizing requirements in academia. Most commonly, requirements are categorized into functional requirements (FRs) and non-functional requirements (NFRs). However, as pointed out by Glinz and Broy, there is a terminological confusion about the underlying terms.

This disagreement over categorizing requirements is not only prevalent in research, but it also influences how requirements are elicited, documented, and validated in practice. As a matter of fact, up until now, there does not exist a commonly accepted approach for the NFR-specific elicitation, documentation, and analysis; so-called NFRs are usually described vaguely, remain often not quantified, and as a result remain difficult to analyze and test. Furthermore, so-called NFRs are often retrofitted in the development process or pursued in parallel with, but separately from functional requirements and, thus, are implicitly managed with little or no consequence analysis. This limited focus on so-called NFRs can result in the long run in high maintenance costs.

Although the importance of so-called NFRs for software and systems development is widely accepted, the discourse in academia is still dominated by how to categorize requirements. One point of view is that a categorization should be grounded in methodological reasons and we should rather base a requirements categorization on a system model. The underlying argument is that if we base a categorization on a system model, we can precisely specify requirements in terms of properties of systems, where properties are represented by logical predicates.

The goal of the dissertation is to address the following two major questions: First, is a requirements categorization based on a system model adequate for requirements found in practice? With adequate, we mean that the categorization is applicable for industrial requirements and supports subsequent development activities. Second, how can a requirements categorization based on a system model be operationalized for subsequent development activities? To this end, we contribute a detailed analysis how practitioners handle and categorize requirements, elaborate the reasons for and resulting consequences on the overall development process, analyze the adequacy of a categorization based on a system model, analyze problems with requirements categorizations in practice, and present and evaluate an approach that embeds such a categorization for requirements in subsequent development activities.


requirements
quality
process
documentation
nonfunctional
categories
research
design
notion
elements
practice
attributes
performance
respondents
activity
activities
analyzed
qrs
logical
state
input
completeness
scope
view
test
categorization
functional
categorizations
practitioners
systems
software
interface
analysis
literature
result
dissertation
specification
availability
frs
types
user
context
distinction
properties
consequences
model
focus
sentence
information
definition
property
explicit
system
function
product
data
probabilistic
probability
behavioral
throughput
capacity
specifications
theory
time
problems
classes
requirement
set
approach
attribute
patterns
component
development
content
behavior
problem
engineering
architecture
study
classification

Für Petra und Martin

Petra und Martin, Danke für eure immerwährende Unterstützung, Danke für eure Liebe, Danke dafür, dass ihr mir immer ein Halt auf meinem Weg seid. Ihr habt mir diese Dissertation überhaupt erst ermöglicht!

Außerdem möchte ich meinem *großen* Bruder Lukas und meiner *kleinen* Cousine Anna danken. Ihr beide erweist immer wieder einen sechsten Sinn und führt mich mit euren Ratschlägen stets wieder auf den rechten Pfad — nicht unbedingt wissenschaftlich, aber im Leben unabdingbar. Ich möchte außerdem Stephanie danken. Danke, dass du mich auf meinem Weg mit Rat und Tat unterstützt hast. Schließlich möchte ich noch Mai 🐌 danken; Du hast mir immer dann ein Lächeln geschenkt wenn ich es am meisten brauchte. Danke!

"Nicht die Glücklichen sind dankbar. Es sind die Dankbaren, die glücklich sind."

— Sir Francis Bacon[a]

[a] Mmmm. . . Bacon (Homer Jay Simpson)

Acknowledgements

Though words cannot express my gratitude, it is a pleasure to thank those who made my dissertation possible; It is your support, friendship, and encouragement that was indispensable for the successful completion of my dissertation.

I would like to sincerely thank Prof. Dr. Dr. h.c. Manfred Broy who always supported me and gave me the challenge to find and opportunity to work on my own topics and interests. I am very grateful for your encouraging confidence in me and for the opportunity to participate in various challenging research projects. I always felt that I got the support I needed to find my own way. I also would like to express my gratitude to my co-supervisor Prof. Dr. Klaus Pohl for stimulating discussions and valuable feedback on the many-faceted topic of my dissertation. Moreover, I would like to thank Prof. Dr. Martin Glinz for inspiring and helpful discussions about requirements categorizations and for his feedback on my work.

I owe my deepest gratitude to Andreas Vogelsang. Andreas, thank you for pushing me into my topic and continuously supporting me. Your support was one of the cornerstones of my dissertation; Thank you for being patient with me—no matter how often I rushed into your office and distracted you. Moreover, I would like to thank Daniel Méndez Fernández. Daniel, as a colleague, you always helped with scientific advice and support; As a friend, you encouraged and helped me with your positive attitude when I most needed it. ¡Muchas gracias!

My sincere thanks go to many friends and colleagues for scientific discussions, advices, and continuous support. Among them, particularly Henning Femmer, helped me with many ideas and good advices during my time at the chair. Henning, thank you for your support and friendship; You taught me how important it is to focus on one topic at a time and that we must go if the mountains are calling. Vasileios Koutsoumpas, I would like to thank you for many inspiring discussions about Greek etymology, philosophy, and mythical fuzzy sets. I cannot imagine a better office partner. Ευχαριστώ. Jakub Mikołaj Mund, thank you for our friendship and also for our inspiring discussions about quality. You taught me that there is no such thing as *just a last one*. Maximilian Junker, thank you explaining probability theory and the probabilistic extension of Focus to me, for helping me with applying my approach to availability, and for being such a critical reviewer. Moreover, I would like to thank Diego Marmsoler for critically reviewing and discussing parts of my formalization. I would also like to thank Benedikt Hauptmann for teaching me to cite the right person right. Moreover, I would like to thank Florian Grigoleit, Ilias Gerostathopoulos, Tanmaya Mahapatra, Georgios Pipelidis, Daniela Steidl, Annabelle Klarl, Veronika Bauer, Thomas Kofler, Wolfgang Böhm, Birgit Penzenstadler, Marco Kuhrmann, Alarico Campetelli, Sebastian Eder, Franz Huber, Dieter Mletzko, Monika Glashauser, Thomas Fritz, Andrea Heller, and Hannelore Plagge for their support and friendship. Last, but not least, I would like to thank Silke Müller—the heart of our chair—who supported me in so many ways.

Contents

"If you want to trigger a hot debate among a group of requirements engineering people, just let them talk about non-functional requirements."

— Martin Glinz, 2007

1 Chapter Introduction

This dissertation is about requirements categorizations in general, and in particular about the adequacy and operationalization of a categorization based on a system model. In this dissertation, with *adequacy of a requirements categorization*, we mean that the categorization is applicable for industrial requirements and, furthermore, supports subsequent development activities. In this chapter, we introduce and motive this topic by discussing categorizations and the associated consequences in practice (Section 1.1). In Section 1.2 we introduce our research hypothesis and formulate the problem statements. We derive the research objectives in Section 1.3 and present the overall research methodology and major contributions in Section 1.4.

1.1. Context: Requirements Categorizations and their Consequences in Practice

Requirements Engineering (RE) is an essential part of every software and systems engineering project. It can be defined as *the process of discovering the purpose of software systems by identifying stakeholders and their needs and by documenting these in a form that is amenable to analysis, communication, and subsequent implementation* [Nuseibeh and Easterbrook, 2000]. RE has a crucial impact on the functionality, quality, costs and, thus, the usefulness and resulting complexity of the system under consideration. Many problems arise if RE is underestimated or conducted thoughtlessly; Important requirements are often not identified and, thus, essential requirements are not fulfilled by the final product. Moreover, the later an error is found, the higher are the associated costs for revising it [Boehm, 1981; Mund et al., 2015]. In particular, Boehm estimates

that an error in the requirements discovered while implementing a system, needs about 20 times more effort to correct compared to when it is discovered in the requirements engineering stage; If the error is detected after delivering the software, an effort of a factor of 100 is needed [Boehm, 1981].

1.1.1. Problems of Current Categorizations

There is a significant academic discussion on categorizing requirements. Most commonly [IEEE Std 830-1998, 1998; Pohl, 2010; Robertson and Robertson, 2012; Sommerville and Kotonya, 1998; Sommerville and Sawyer, 1997; Van Lamsweerde, 2001], requirements are categorized into *functional requirements (FRs)* and *non-functional requirements (NFRs)*. However, as pointed out by e.g. Pohl [2010], Glinz [2007], and Broy [2016], there is a terminological confusion about the underlying terms. Glinz points out three major problems with current categorizations of non-functional requirements, including the so-called *definition problem*. He discusses that there is no consensus about the terms used by current definitions of non-functional requirements. Moreover, Broy [2016] points out that there is no agreed meaning of the terms *functional* and *non-functional*. The term *functional* may be understood in multiple ways. For example, in the mathematical way, i.e., a mapping between two sets, or in an engineering way, i.e., the purpose of a system or of a certain part thereof. Similarly for the term *non-functional*; The obvious way to categorize non-functional requirements as all those requirements that are *not* functional is overly simplistic and does not provide a helpful and actionable categorization. Pohl [2010] argues that the term "non-functional requirement" should not be used as they are essentially underspecified requirements and we should rather categorize requirements into *functional requirements*, *quality requirements (QRs)*, and *constraints*.

Still, the structuring principles used for most approaches are not completely clear, not overly precise, and not explicitly stated [Broy, 2016]. For example, Pohl [2010], the IEEE Std 830-1998 [1998], Lauesen [2002], Robertson and Robertson [2012], Sommerville [2007], and Wiegers and Beatty [2013], categorize requirements in *functional requirements*, *quality requirements*, and *constraints*. In their categorization, a quality requirement is defined as *"a quality property of the entire system or of a system component, service, or function"* (see e.g. [Pohl, 2010]). If we analyze this definition in detail, we can see that the scope of a quality requirement is specified as the *"entire system"*, a *"system component"*, a *"service"*, or a *"function"*. Yet, in the second part of the definition, the equally fuzzy term *"quality property"* is used. Pohl [2010], for example, further gives examples and a short definition for quality properties like *"availability refers to the percentage of time during which the system is actually available for use and fully operational"*. However, again, fuzzy and not clearly defined terms like *"actually available"* and *"fully operational"* are used. Thus, we argue that the main principles of requirements categorizations are not clearly and precisely defined in literature and thus leaves room for interpretation. This is not only an academic problem—and even more importantly—this confusion impacts how requirements and in particular QRs are handled in practice.

1.1.2. Resulting Consequences in Practice

This disagreement with requirements categorizations is not only prevalent in research, but it also influences how requirements are elicited, documented, and validated in practice [Ameller et al., 2012; Borg et al., 2003; Chung and Nixon, 1995; Svensson et al., 2009]. As a matter of fact, up until now, there does not exist a commonly accepted approach for the QR-specific elicitation, documentation, and analysis [Borg et al., 2003; Svensson et al., 2009]; QRs are usually described vaguely [Ameller et al., 2012; Borg et al., 2003], remain often not quantified [Svensson et al., 2009], and as a result remain difficult to analyze and test [Ameller et al., 2012; Borg et al., 2003; Svensson et al., 2009]. Furthermore, QRs are often retrofitted in the development process or pursued in parallel with, but separately from, functional requirements [Chung and Nixon, 1995] and, thus, are implicitly managed with little or no consequence analysis [Svensson et al., 2009]. It is an open question whether these problems are caused by the nature of QRs or by the mere fact that a requirement is marked as such. Still, this limited focus on QRs can result in the long run in high maintenance costs [Svensson et al., 2009].

In particular, QRs like reliability, security, or performance strongly influence the architecture of a system. For example, a security requirement like *"only authenticated and authorized users are allowed to access the software"* demands a mechanism for user authentication, authorization, and access rights from the software. If this requirement is not fulfilled, the overall project success may be in danger. Moreover, a performance requirement specifying maximal bounds on latencies for a distributed telecommunication software, requires a high effort to test, as a test setup with a large amount of participants in the network is needed. Furthermore, the usability requirement *"the time-based zoom function must be comfortable to use"* describes an external property of the system in relation to its usage. It has a strong impact on the user interface and the interactions of the user with the software. Because of its inherent subjectivity, it is further hard and time-consuming to test.

In summary, there is a disagreement with requirements categorization in academia, which influences how requirements are handled in practice. Still, it is widely acknowledged that QRs are vital for the success of software systems in practice [Chung and Nixon, 1995], although there are many issues with how QRs are handled in practice.

1.2. Research Hypothesis & Problem Statement

Although the importance of QRs for software and systems development is widely accepted, the discourse in academia is dominated by how to categorize requirements [Broy, 2016; Glinz, 2007]. One point of view is that a categorization should be unambiguous and grounded in methodological reasons. Therefore, we should rather base a requirements categorization on a system model that provides us with a clear notion and concept of a system. The underlying argument is that if we base a categorization on a system model, we can precisely specify requirements in terms of properties of systems, where properties are represented by logical predicates. This allows us to precisely and explicitly specify the structuring principles of the categorization. For example, Broy [2015, 2016] catego-

rizes requirements in *behavioral properties* and *representational properties*. Behavioral properties subsume traditional functional requirements, such as *"the user must be able to remove articles from the shopping basket"* as well as so-called QRs that describe behavior such as *"the system must react on every input within 10ms"*. Representational properties include so-called QRs or constraints that determine how a system shall be syntactically or technically represented, such as *"the software must be implemented in the programming language Java"* [Broy, 2015, 2016].

Furthermore, given a requirements categorization that is based on a system model, the seamless transition to architectural design (operationalization) is facilitated, as requirements are built on clearly defined and explicitly stated logical properties over a set of systems. Following these arguments, we formulate the research hypothesis for this dissertation:

> **Research Hypothesis:**
> A requirements categorization based on a system model is adequate[1] for requirements found in practice and is operationalizable for subsequent development activities.

Based on this hypothesis, we derive two problem statements that we aim to address in this dissertation. First, we want to address the first part of the hypothesis, i.e., is a categorization based on a system model adequate for requirements found in practice? Second, we want to propose an approach to embed and operationalize the categorization in subsequent development activities. More precisely, we derive the following two problem statements:

> **Problem Statement:**
>
> **P1:** Is a requirements categorization based on a system model adequate[1] for requirements found in practice?
>
> **P2:** How can a requirements categorization based on a system model be operationalized for subsequent development activities?

1.3. Research Objectives

Based on the two problem statements, we formulate the following objectives, which we pursue in the remainder of this dissertation:

Objective 1: *We want to understand how and why practitioners categorize requirements and what are the resulting consequences.* In this objective, we aim to understand the state of the practice. In particular, we want to understand how practitioners

[1] In this dissertation, with *adequacy of a requirements categorization*, we mean that the categorization is applicable for industrial requirements and, furthermore, supports subsequent development activities.

handle requirements, what are the reasons for and what are the consequences of the way they handle requirements.

Objective 2: *We want to assess if a categorization based on a system model is adequate[1] for requirements found in practice and whether it effectively supports subsequent development activities.* In this objective, we focus on requirements found in practice. In particular, we want to analyze whether a categorization based on a system model is adequate for those requirements and, furthermore, want to discuss whether such a categorization effectively supports subsequent development activities.

Objective 3: *We want to develop an approach that is based on such a categorization and want to assess whether it is applicable in practice.* In this objective, we aim to propose an approach that embeds and operationalizes a categorization that is based on a system model in subsequent development activities. In particular, we aim to propose an approach that allows practitioners to specify QRs based on clearly defined concepts.

1.4. Research Methodology & Contributions

In this dissertation, we provide supporting evidence for our hypothesis and solutions for the stated problems. In particular, this dissertation analyzes in detail how practitioners categorize and handle requirements, elaborates the reasons for and resulting consequences on the overall development process, analyses the adequacy of a categorization based on a system model, analyzes and critically discusses the implications of requirements categorizations in practice, and presents and evaluates an approach that embeds such a categorization for requirements in subsequent development activities. Figure 1.1 shows an overview of the research methodology and contributions of this dissertation; Empirical studies are marked with an E, constructive contributions with a C, and validation studies with a V. The dissertation presents the following main contributions to the current state of the art:

Contribution 1 *An investigation of how practitioners categorize and handle requirements (E).* In this contribution, we report on a survey we conducted with 109 practitioners to explore whether and, if so, why they consider requirements labeled as FRs differently from those labeled as QRs as well as to disclose resulting consequences for the development process.

88% of our respondents stated that they document QRs and 85% of them declare to make an explicit distinction between QRs and FRs. Furthermore, our results indicate that the development process strongly differs between QRs and FRs, especially in late phases such as testing. This was true regardless of the style of documentation. We further identified a number of reasons why practitioners tend to (or not to) distinguish between QRs and FRs, and we analyzed both problems and benefits from distinguishing (or not) between QRs and FRs. The reported reasons include: QRs have a different nature, are cross-functional, strongly influence the

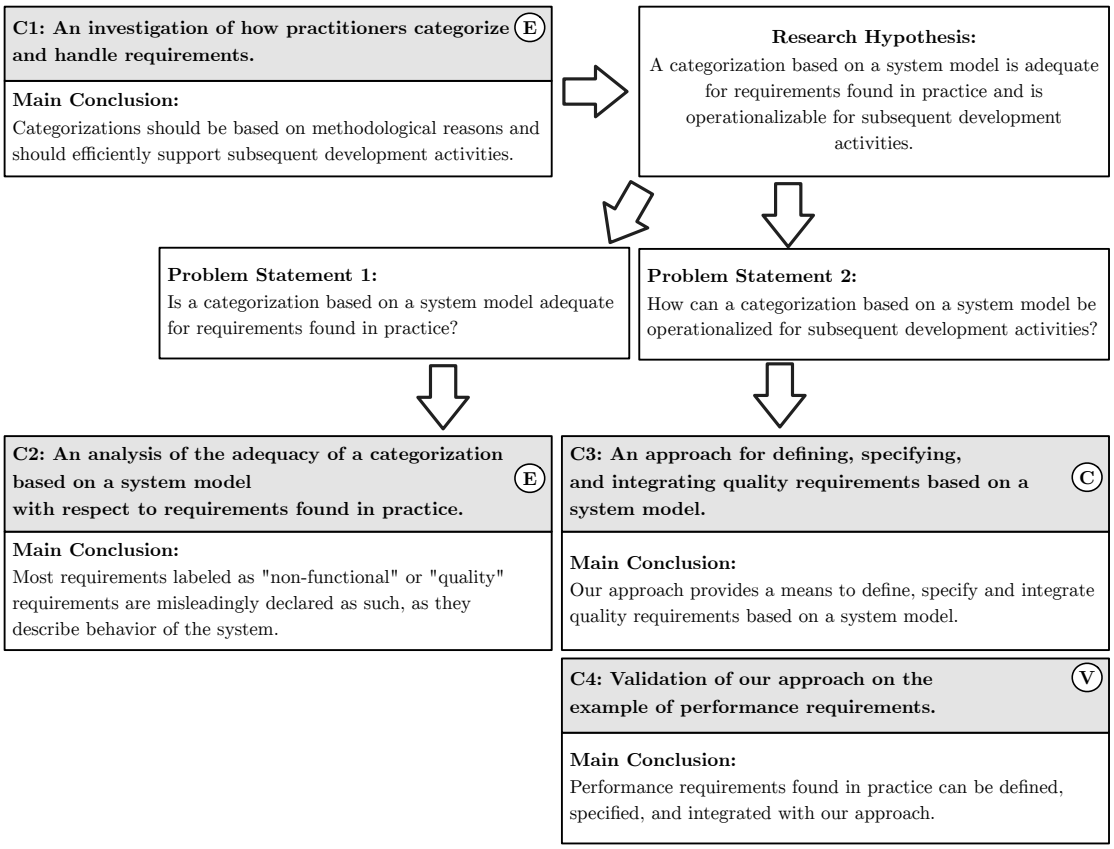


Figure 1.1.: Overview of the research methodology and contributions of this dissertation. Empirical studies are marked with an **E**, constructive contributions with a **C**, and validation studies with a **V**.

architecture, and require different verification methods. Moreover, another reason for making a distinction was reported to lie in company practices or the company processes. We also identified problems that result from this distinction such as that traceability becomes more expensive, the distinction between QRs and FRs is blurry, and QRs are neglected or forgotten along the process.

Our results indicate that making a distinction or not does not have negative or positive consequences per se. It therefore seems more important that the decision whether to make an explicit distinction or not should be made consciously such that people are also aware of the risks that this distinction bears so that they may take appropriate countermeasures. A distinction might, for example, be justified by specialized testing teams for specific quality attributes or by requirements that are reused between a number of projects. A direct consequence of this conscious decision is that people are also aware of the potential risks that this distinction bears (e.g., the importance of trace links between FRs and QRs to assure that QRs are not neglected). In summary, we conclude that QRs are not (sufficiently) integrated in the software development process and furthermore that several problems are evident with QRs.

Contribution 2 *An analysis of the adequacy of a categorization based on a system model with respect to requirements found in practice (E).* In this contribution, we analyze 11 requirements specifications from 5 different companies for different application domains and of different sizes with in total 530 requirements that are labeled as "non-functional", "quality", or any specific quality attribute. Our results show that 75% of the requirements labeled as "quality" in the considered industrial specifications describe system behavior and 25% describe the representation of the system. As behavior has many facets, we further categorize behavioral QRs according to the *system view* they address (interface, architecture, or state), and the *behavior theory* used to express them (syntactic, logical, probabilistic, or timed) [Broy, 2015, 2016]. Based on this fine-grained categorization that is based on a system model, we discuss the implications we see on handling QRs in the software development phases, e.g., testing or design.

Based on these results, we argue that functional requirements describe any kind of behavior over the interface of the system, including timing and/or probabilistic behavior. From this perspective, we conclude that many of those QRs that address system properties describe the same type of behavior as functional requirements do. This is true for almost all QR classes we analyzed; even for QR classes which are sometimes called *internal* quality attributes (e.g., portability or maintainability) [McConnell, 2004]. Hence, we argue that Broy's requirements categorization—that is based on a system model—is adequate[1] for requirements found in practice, as the categories can be linked to system development activities. From a practical point of view, this means that most QRs can be elicited, specified, and analyzed like functional requirements. For example, QRs classified as black-box interface requirements, are candidates for system tests. In our data set, system test

cases could have been specified for almost 51.5% of the QRs. This contribution supports (the first part of) our hypothesis, i.e., a categorization based on a system model is adequate[1] for requirements found in practice.

Contribution 3 *An approach for defining, specifying, and integrating quality requirements based on a system model (C).* In this contribution, we present an approach for defining, specifying, and integrating QRs based on a system model. In particular, the approach takes a specific quality attribute as input and creates a precise and explicit definition and customized sentence patterns for requirements concerning this quality attribute. We achieve the precise and explicit definition by an explicit mapping to a system model and the customization to a given organizational context by using the idea of activity-based quality models [Deissenboeck et al., 2007; Femmer et al., 2015]. Furthermore, we give guidance how the individual steps can be performed by means of executing the approach for the quality attributes performance and availability.

The resulting definitions and sentence patterns can then be integrated in the overall RE process to support the documentation, elicitation, management, and validation of requirements in the given organizational context. Thus, our approach needs to be conducted in advance for a given set of quality requirements and a given context; The results can then be used as for example a company standard to specify and elicit quality requirements.

This contribution supports (the second part of) our hypothesis, i.e., it provides an approach for an operationalization of the categorization for subsequent development activities.

Contribution 4 *Validation of our approach on the example of performance requirements (V).* In this contribution, we instantiate our approach for one specific quality attribute (performance requirements) and conduct an evaluation with respect to its applicability and ability to uncover incompleteness.

In particular, we present a context-independent and context-dependent content model for performance requirements, a clear and precise definition of the content elements and a discussion how to express them based on the Focus system model, and an operationalization through sentence patterns for the specification of performance requirement. Furthermore, we derive a notion of completeness based on the context-dependent content model. To make the model widely applicable, we based the context-independent content model on broad categorizations of non-functional/ quality requirements in literature [Behkamal et al., 2009; Boehm et al., 1976; Botella et al., 2004; Dromey, 1995; Glinz, 2005, 2007; Grady, 1992; ISO/IEC 25010-2011, 2011; ISO/IEC 9126-2001, 2001; McCall et al., 1977; Robertson and Robertson, 2012; Sommerville, 2007], unifying the different aspects of performance described in the individual categorizations. To make our approach applicable in practice, we operationalize the content model through sentence patterns for performance requirements. To evaluate our approach, we applied the resulting sentence patterns

to 58 performance requirements taken from 11 industrial specifications and analyzed (i) the applicability and (ii) the ability to uncover incompleteness. We were able to rephrase 86% of the performance requirements. Moreover, we found that the resulting sentence patterns can be used to detect incompleteness in performance requirements, revealing that 68% of the analyzed performance requirements were incomplete.

This contribution supports (the second part of) our hypothesis, i.e., it provides an instantiation and assessment of the approach.

Delimitation: Focus on Product-related Requirements

We focus on **product-related** requirements, i.e., requirements that describe properties of the product or system under development, and explicitly exclude **process-related requirements**, i.e., requirements that describe properties concerning the development process[2]. We further categorize product-related requirements into *functional requirements*, *quality requirements*, and *constraints*. Moreover, in contrast to e.g. Pohl [2010], we do not understand "non-functional" requirements as underspecified requirements. We do consider functional requirements and quality requirements without considering the level of underspecification of the requirement. Thus, for the remainder of this dissertation, the terms *product-related non-functional requirement* and *quality requirement* only differ with respect to one point: quality requirements additionally provide the quality property to which they refer, while non-functional requirements do not.

1.5. Outline

This dissertation consists of ten chapters. Chapter 2 (Fundamentals) introduces the fundamentals of this dissertation: the requirements engineering discipline and its important concepts, the Focus theory and its probabilistic extension, and Broy's requirements categorization (which is based on the Focus system model). Chapter 3 (Related Work) describes the current state of the art regarding the contents of this dissertation in general. In particular, we report on related work regarding requirements categorizations and their implications in practice. Chapter 4 (An Investigation of How Practitioners Handle Requirements) investigates how practitioners categorize and handle requirements. In Chapter 5 (An Analysis of the Adequacy of a Categorization based on a System Model with Respect to Requirements found in Practice), we empirically investigate whether a requirements categorization that is based on a system model is adequate[1] for industrial requirements. Chapter 6 (An Analysis of Requirements Categorizations and their Consequences in Practice) analyzes the state of the practice and derives problems with requirements categorizations sketches possible solutions to overcome the deficiencies

[2]Sometimes, the distinction between product and process-related requirements is not made and process-related requirements are subsumed as constraints. In this dissertation, we want to make explicit that we focus on product-related requirements.

associated with QR in practice. In Chapter 7 (An Approach for Defining, Specifying, and Integrating Quality Requirements based on a System Model), we introduce our approach for defining, specifying, and integrating quality requirements and exemplarily show how to apply it for performance and availability requirements. Chapter 8 (Validation of the Approach on the Example of Performance Requirements) evaluates our approach with respect to its applicability and ability to detect incompleteness in industrial performance requirements. In Chapter 9 (Reflection on the Expressiveness of our Approach), we discuss the limitations of our approach and, finally, in Chapter 10 (Conclusions & Outlook), we conclude the dissertation and formulate future research directions.

Previously Published Material

Parts of this dissertation have been previously published in the following publications:

- Eckhardt, J., Méndez Fernández, D., and Vogelsang, A. (2015). How to specify Non-functional Requirements to support seamless modeling? A Study Design and Preliminary Results. In *Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 164–167 (short paper, research track, 4 pages)
- Eckhardt, J., Vogelsang, A., and Méndez Fernández, D. (2016c). Are Non-functional Requirements Really Non-functional? An Investigation of Non-functional Requirements in Practice. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 832–842 (full paper, research track, 10 pages)
- Eckhardt, J., Vogelsang, A., Femmer, H., and Mager, P. (2016b). Challenging Incompleteness of Performance Requirements by Sentence Patterns. In *Proceedings of the 24th International Requirements Engineering Conference (RE)*, pages 46–55 (full paper, research track, 10 pages)
- Eckhardt, J., Vogelsang, A., and Mendéz Fernández, D. (2016d). On the Distinction of Functional and Quality Requirements in Practice. In *Proceedings of the 17th International Conference on Product-Focused Software Process Improvement (PROFES)*, pages 31–47 (full paper, research track, 16 pages)
- Eckhardt, J., Vogelsang, A., and Femmer, H. (2016a). An Approach for Creating Sentence Patterns for Quality Requirements. In *Proceedings of the 6th International Workshop on Requirements Patterns (RePa)*, pages 308–315 (full paper, 8 pages)

"We are like dwarfs on the shoulders of giants, so that we can see more than they, and things at a greater distance, not by virtue of any sharpness of sight on our part, or any physical distinction, but because we are carried high and raised up by their giant size."

— Bernard De Chartres, 12th century

Chapter 2
Fundamentals

The goal of this chapter is to provide the fundamentals for this dissertation. First, to set the scope of this dissertation, in Section 2.1, we introduce fundamentals on requirements engineering in general, including its major goals, its core concepts and definitions. As this thesis discusses a categorization based on a system model, we provide an overview of a particular system modeling theory, namely the Focus theory and its probabilistic extension in Section 2.2. Finally, in Section 2.3, we introduce Broy's requirements categorization that is based the system model of the Focus theory.

2.1. Fundamentals: Requirements Engineering

The IEEE standard committee defines *software engineering* as a systematic and cost-effective approach to software development projects [IEEE Std 610.12-1990, 1990], which can be divided into different phases, such as requirements engineering, design, implementation or testing. In this dissertation, we are in particular interested in requirements engineering (RE). To this end, we subsequently introduce the foundations in the area of requirements engineering.

2.1.1. Goals of Requirements Engineering

RE aims to *discover the purpose of software systems*, *identify stakeholders and their needs*, and *document these in a form that is amenable to analysis, communication, and subsequent implementation* [Nuseibeh and Easterbrook, 2000]. According to Zave [1997], RE has the following goals:

G1 To capture and specify the *problem space*, i.e., "real-world goals for functions and constraints on software systems" [Zave, 1997].

G2 To continuously administrate the specified requirements (and changes) over their whole life cycle.

Thus, the goals are twofold: on the one-hand, the goal is to capture and specify the problem space and, on the other-hand, to manage the resulting artifacts.

2.1.2. Requirements Engineering

Requirements Engineering (RE) can be defined as the activity of "describing a problem space as comprehensively as possible by comprising iterative and systematic approaches to define a requirements specification aligned to the needs of all relevant stakeholders" [Méndez Fernández, 2011].

We base our understanding of RE on the definition by Pohl [2010]. Pohl defines three dimensions of requirements engineering, in particular, the *content dimension*, the *agreement dimension*, and the *documentation dimension*. The content dimension ensures that all requirements are known and understood in detail, the agreement dimension considers the establishment of sufficient stakeholder agreement, and the documentation dimension considers the documentation/specification of requirements in compliance with the defined formats and rules. Based on these three dimensions, he defines requirements engineering as a process:

Requirements Engineering (Process) Requirements engineering is a cooperative, iterative, and incremental process which aims at ensuring that:

- All relevant requirements are explicitly known and understood at the required level of detail.
- A sufficient agreement about the system requirements is achieved between the stakeholders involved.
- All requirements are documented and specified in compliance with the relevant documentation/specification formats and rules.

Note. *We additionally follow the view of, e.g., Femmer et al. [2015], as we understand RE as a supporting means for software engineering, with the goal to produce working software products in a systematic and predictable way. Therefore, the value of the outcome of RE cannot be assessed on its own but must be evaluated in its use as a function to the rest of the engineering endeavor.*

Requirements Engineering Activities Pohl [2010] derives three core activities that significantly contribute to the achievement of the three goals stated above:

Documentation: The focus of this activity is the documentation and specification of the elicited requirements and other important information according to defined documentation and specification rules.

Elicitation: The focus of this activity is to improve the understanding of the requirements. In particular, during the elicitation activity, requirements are elicited from stakeholders and other requirements sources, e.g., laws and standards, and the requirements are collaboratively developed.

Negotiation: The focus of this activity is to detect, explicitly state, and resolve all conflicts between the viewpoints of the different stakeholders.

Pohl further defines two cross-sectional activities that significantly influence the requirements engineering process. They support the core activities and secure the results of the overall requirements engineering.

Validation: The focus of this activity is to validate the requirements (artifacts), i.e., to detect defects in requirements, to validate the core activities, i.e., to check the compliance between the activities performed and the process and/or activity specifications, and to validate the consideration of the system context, i.e., to validate whether the system context has been considered in the intended way during requirements engineering.

Management: The focus of this activity is the management of the requirements artifacts, the management of the activities, and the observation of the system context.

Artifact-based Requirements Engineering In general, there are two paradigms for establishing an RE approach: *activity orientation* and *artifact orientation*. In an activity-oriented RE approach, a RE reference model is provided as an ordered set of activities and methods, each defining procedures and techniques for a particular purpose, from which project participants can select the appropriate one to design their project-specific RE process [Méndez Fernández and Penzenstadler, 2015]. That is, activity orientation prescribes *how* to do something. In contrast to this, artifact-orientation abstracts from the way of creating the results and specifies *what* has to be done [Méndez Fernández et al., 2010], i.e., project participants concentrate on the RE artifacts rather than on the way of creating them. The research group of Broy takes an artifact-centric perspective on requirements engineering [Méndez Fernández, 2011; Méndez Fernández and Penzenstadler, 2015; Méndez Fernández et al., 2010]; Méndez Fernández and Penzenstadler summarize over six years of experiences in fundamental and evidence-based research on requirements engineering in the AMDiRE approach [Méndez Fernández and Penzenstadler, 2015].

The artifact model constitutes the backbone by defining *structure* and *content* of domain-specific results of RE activities [Méndez Fernández et al., 2010]. The structure addresses the aspects of hierarchically ordered documents or data sets being produced during development tasks in which single content items serve as containers for the concepts, respectively concept models. The concept models (or content model) define those aspects

of a system under consideration dealt with during the development process and reflected in the elements of the description techniques and their relations [Méndez Fernández et al., 2010].

Figure 2.1 shows the AMDiRE artifact model. It contains three artifacts, the *context specification*, the *requirements specification*, and the *system specification*. The context specification defines the context of the system under consideration (SuC) including the specification of the overall project scope, the stakeholders, rules, goals, and constraints as well as a specification of the domain model. The requirements specification comprises the requirements of the SuC, taking a black-box view on the system, i.e., from a user's perspective without constraining the internal realization of the system. Finally, the system specification comprises a glass-box view on the internal realization of a system including a logical component architecture and a specification of the behavior.

Note. *In AMDiRE, the context and requirements specification address the problem space and the system specification addresses the solution space and is the interface to tie in with the design phase (see Section 2.1.6 for a discussion of the problem space and the solution space).*

2.1.3. Requirements

There are several definitions of the term requirement in literature. In the remainder of this dissertation, we understand the term *requirement* as defined in the IEEE Std 610.12-1990 [1990]:

Requirement A requirement is a

1. Condition or capability needed by a user to solve a problem or achieve an objective.

2. Condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.

3. Documented representation of a condition or capability as defined in 1 or 2.

Note. *In this dissertation, we do not distinguish between requirements and requirement artifacts, i.e., the term requirement denotes both, the concept and its representation.*

2.1.4. Requirements Categorizations

In literature, there are mainly two different approaches to categorize requirements: (i) a requirements categorization into *functional* requirements, *non-functional* requirements and (ii) into *functional* requirements, *quality* requirements, and *constraints*.

Some categorizations further distinguish between *product-related* and *process-related* requirements. In this dissertation, we focus on product-related requirements, i.e., requirements that concern properties of the product or system under development.

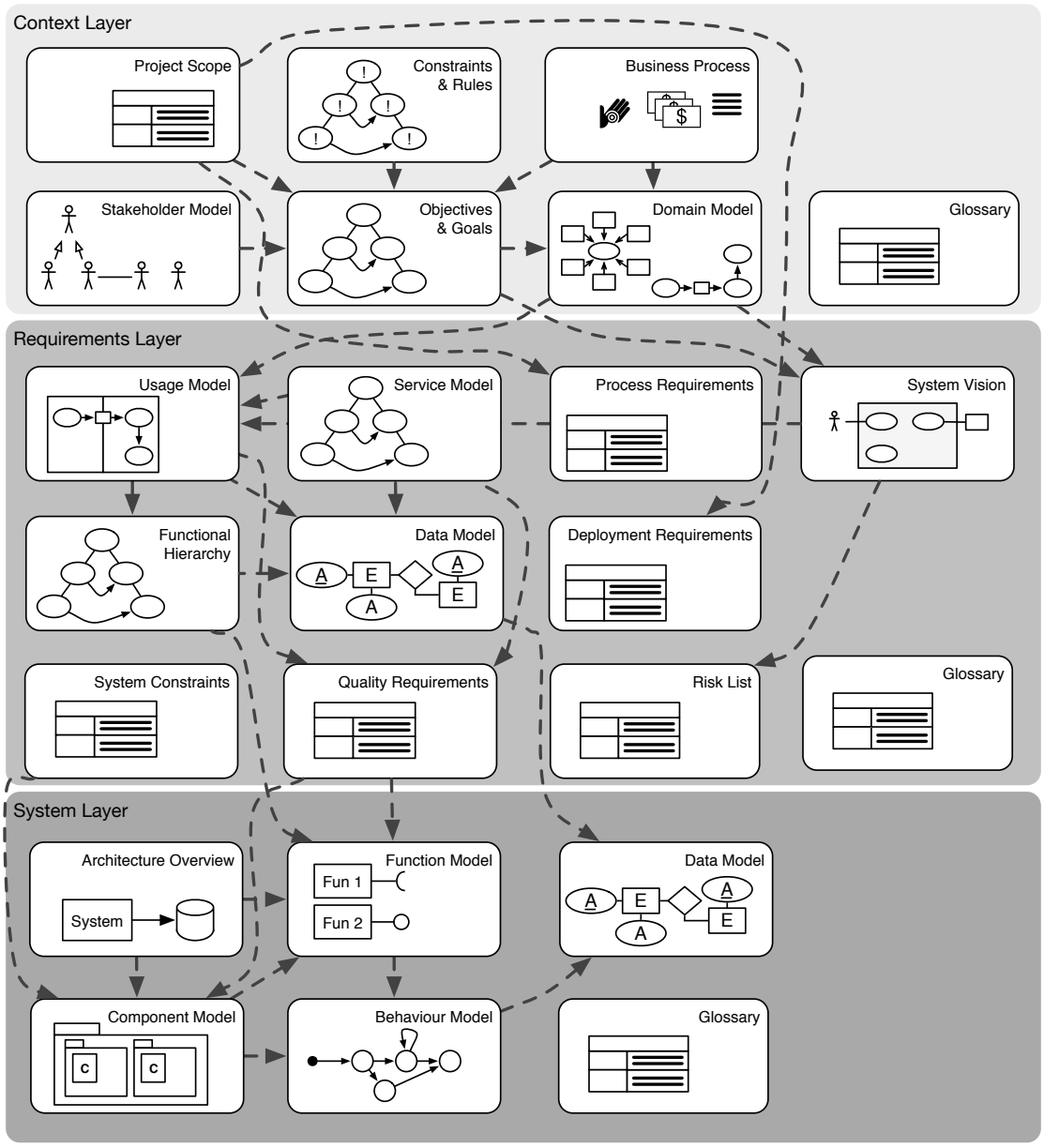


Figure 2.1.: Overview of AMDiRE artifact model.

Functional vs. Non-functional Requirements

Many authors categorize requirements into *functional* and *non-functional* [Antón, 1997; Davis, 1993; Glinz, 2007; Jacobson et al., 1999; Landes and Studer, 1995; Mylopoulos et al., 1992; Ncube, 2000; Paech and Kerkow, 2004; Robertson and Robertson, 2012; Sommerville and Kotonya, 1998; Van Lamsweerde, 2001; Wiegers and Beatty, 2013; Young, 2004]. There is more or less a consensus about the term functional requirement[3]. Usually, a functional requirement expresses that a system offers a certain behavior such that it can be used for a specific purpose. For example, Sommerville [Sommerville, 2007] defines functional requirements as follows.

Functional Requirements "These [functional requirements] are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also state what the system should not do. [...] When expressed as user requirements, the requirements are usually described in a fairly abstract way. However, functional system requirements describe the system function in detail, its inputs and outputs, exceptions, and so on."[4]

Non-functional Requirements For the term *non-functional requirement*, there is no consensus in literature. Table 2.1 gives an overview of common definitions of the term.

Glinz [2007] analyzed most of the classifications and derived three major problems with the notion of non-functional requirements: *the definition problem*, *the classification problem*, and the *representation problem*. The definition problem pinpoints the terminological and conceptual discrepancies in the various definitions, the classification problem pinpoints the differences in the concepts for sub-classifying non-functional requirements, and the representation problem pinpoints that the notion of a non-functional requirement is representation dependent. In summary, there is no consensus about the definition of the term non-functional requirement in literature, the concepts are fuzzy, and representation dependent.

Note. *Pohl [2010] argues that non-functional requirements are in fact underspecified requirements that allow many different interpretations concerning the desired system properties. He argues that non-functional requirements should be refined to functional requirements or quality requirements.*

Categorization based on (Software) Quality Models

There are several categorizations [Chung et al., 2012; ISO/IEC 25010-2011, 2011; ISO/IEC 9126-2001, 2001; Lauesen, 2002; Lochmann and Wagner, 2012; Wiegers and Beatty, 2013]

[3] Broy [Broy, 2016] points out that there is no agreed meaning of the term *function*. It may be understood in multiple ways; For example in the mathematical way, i.e., a mapping between two sets, or in an engineering way, i.e., the purpose of a system or of a certain part thereof.

[4] The reported definition is adapted from the definition presented in [Pohl, 2010].

Table 2.1.: Definitions of the term "non-functional requirement". Adapted and extended from Glinz [2007].

Source	Definition
Antón [1997]	Describe the **non-behavioral aspects** of a system, capturing the **properties** and **constraints** under which a system must operate.
Davis [1993]	The required overall **attributes** of the system, including portability, reliability, efficiency, human engineering, testability, understandability, and modifiability.
Jacobson et al. [1999]	A requirement that specifies **system properties**, such as **environmental and implementation constraints**, performance, platform dependencies, maintainability, extensibility, and reliability. A requirement that specifies **physical constraints** on a functional requirement.
Sommerville and Kotonya [1998]	Requirements which are not specifically concerned with the **functionality** of a system. They place **restrictions** on the product being developed and the development process, and they specify **external constraints** that the product must meet.
Mylopoulos et al. [1992]	[...] global requirements on its development or operational cost, performance, reliability, maintainability, portability, robustness, and the like. [...] There is not a formal definition or a complete list of nonfunctional requirements.
Ncube [2000]	The **behavioral properties** that the specified functions must have, such as performance, usability.
Robertson and Robertson [2012]	A **property**, or **quality**, that the product must have, such as an appearance, or a speed or accuracy property.
Wiegers and Beatty [2013]	A description of a **property** or **characteristic** that a software system must exhibit or a **constraint** that it must respect, other than an observable **system behavior**.
Glinz [2007]	A non-functional requirement is an **attribute** of or a **constraint** on a system.
Landes and Studer [1995]	NFRs constitute the justifications of design decisions and **constrain** the way in which the required **functionality** may be realized.
Paech and Kerkow [2004]	The term *non-functional requirement* is used to delineate requirements focusing on *how good* software does something as opposed to the functional requirements, which focus on *what* the software does.
Van Lamsweerde [2001]	[...] types of concerns: functional concerns associated with the services to be provided, and nonfunctional concerns associated with **quality of service** such as safety, security, accuracy, performance, and so forth.
Young [2004]	A necessary **attribute** in a system that specifies how functions are to be performed, often referred to in systems engineering as the *-ilities*.

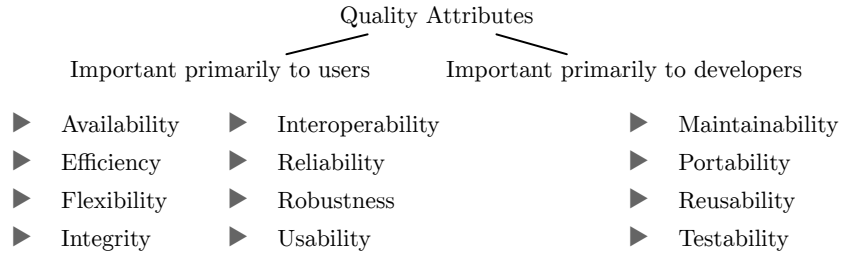


Figure 2.2.: Software Quality Model as defined by Wiegers and Beatty [2013] adapted from Pohl [2010].

that categorize requirements into *functional* requirements, *quality* requirements, and *constraints* with a similar definition for the term "functional requirement" as introduced above. However, they advise against the usage of the term "non-functional" as they argue that non-functional requirements are in their essence underspecified functional or quality requirements [Pohl, 2010]. Instead, they introduce the category *quality requirements*. A quality requirement is then defined as requirement describing a quality property of a system with respect to a quality model. Quality requirements then describe a quality property of the system, while quality properties are defined based on (software) quality models. Wiegers and Beatty [2013] define such a quality model. He distinguishes between quality attributes *important primarily to users* and quality attributes *important primarily to developers*. Figure 2.2 gives an overview of his quality model and Table 2.2 gives a short description of the quality characteristics. In addition, Lochmann and Wagner [2012] extend this understanding of quality by the paradigm of activity-based quality models [Deissenboeck, 2009; Deissenboeck et al., 2007]. They define the quality attributes by referring to activities that are conducted with or on the system and identify (intrinsic) properties that influence the activities' efficiency or effectiveness. For example, the quality attribute *maintainability* is defined as the efficiency (i.e., the effort spent) and effectiveness (i.e., the resulting stakeholders' satisfaction) of conducting the activity *maintenance* [Lochmann and Wagner, 2012]. They chose to use activities for decomposing quality as they provide a clear decomposition criterion: activities may be composed into sub-activities. Figure 2.3 shows the structure and the quality attributes of the quality model. The quality model consists of a tree of main quality attributes, a list of auxiliary quality attributes, and a tree of orthogonal quality attributes. An individual definition for the quality attributes can be found in the original work [Lochmann and Wagner, 2012].

Given a quality model, a quality requirement is usually defined as [Pohl, 2010]:

Quality Requirement A quality requirement defines a quality property of the entire system, of a system component, service, or function.

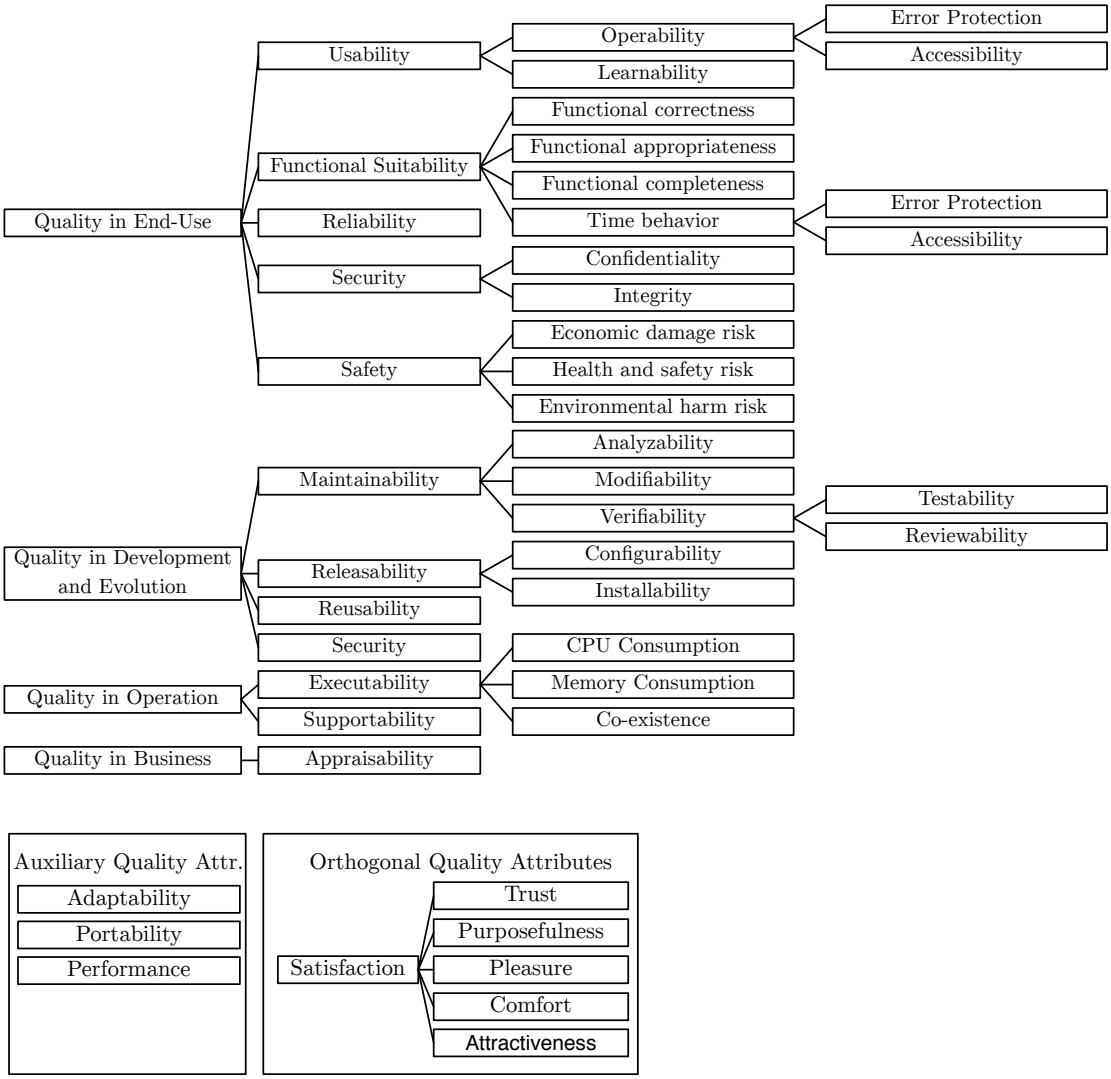


Figure 2.3.: (Activity-based) Software Quality Model as defined by Lochmann and Wagner [2012].

Table 2.2.: Quality attributes defined by Wiegers and Beatty [2013].

Quality Attributes	**Description**
Availability	Availability refers to the percentage of time during which the system is actually available for use and fully operational.
Efficiency	Efficiency is a measure of how well the system utilizes hardware resources such as processor time, memory, or communication bandwidth.
Flexibility	Flexibility indicates how much effort is needed to extend the system with new capabilities.
Integrity	Integrity denotes how well the system is protected against unauthorized access, violations of data privacy, information loss, and infections through maleficent software.
Interoperability	Interoperability indicates how easily the system can exchange data or services with other systems.
Reliability	Reliability is the probability of the system executing without failure for a specific period of time.
Robustness	Robustness is the degree to which a system or component continues to function correctly when confronted with invalid inputs, defects in connected systems or components, or unexpected operating conditions.
Usability	Usability measures the effort the user requires to prepare input for, operate, and interpret the output of the system.
Maintainability	Maintainability indicates how easy it is to correct a defect or make a change in the system.
Portability	Portability relates to the effort it takes to migrate a system or component from one operating environment to another.
Reusability	Reusability indicates the extent to which a component can be used in systems other than the one for which it was initially developed.
Testability	Testability refers to the ease with which the software components or integrated system can be tested to find defects.

Constraints A constraint is an organizational or technological requirement that restricts the way in which the system shall be developed [Pohl, 2010; Robertson and Robertson, 2012].

2.1.5. Our Notion of Requirements for this Dissertation

In this dissertation, we analyze requirements categorizations and their adequacy in practice. For this, we take an observatory point of view and do not prefer a certain requirements categorization. However, in this section, we introduce the terminology that we use in the remainder of this dissertation regarding requirements categorization. In particular, we follow the view of Pohl and Robertson & Robertson and categorize requirements into *functional requirements*, *quality requirements*, and *constraints*. Moreover, we focus on **product-related** requirements, i.e., requirements that describe properties of the product or system under development, and explicitly exclude constraints or so-called process requirements.

In contrast to Pohl [2010], we do not understand "non-functional" requirements as underspecified functional or underspecified quality requirements. We do consider functional requirements and quality requirements without considering the level of underspecification of the requirement. Thus, for the remainder of this dissertation, the terms *product-related non-functional requirement* and *quality requirement* only differ with respect to one point: quality requirements additionally provide the quality property to which they refer, as for example *availability*, while non-functional requirements do not.

Therefore, as we analyze how practitioners handle requirements (Chapter 4) and requirements found in requirements specifications in practice (Chapter 5 and Chapter 8), we understand all those requirements that are labeled as "non-functional", "quality", or any specific quality attribute as quality requirements.

We follow the definition of Broy [2015, 2016] for functional requirements, the definition of Pohl [2010] for quality requirements, and the definition of Robertson and Robertson [2012] for constraints.

Functional Requirements A functional requirement of a system expresses that

- a system shall offer a particular functional feature such that the system can be used for a specific purpose, or
- a function of a system having a particular property—that may be a logical property or a probabilistic one—modeling part of the interface behavior of the system, specified by the interaction between the system and its operational context.

Quality Requirement A quality requirement defines a quality property of the entire system, of a system component, service, or function.

Constraints A constraint is an organizational or technological requirement that restricts the way in which the system shall be developed [Pohl, 2010; Robertson and Robertson, 2012].

2.1.6. Problem Space vs. Solution Space

In requirements engineering, it is important to separate the definition of requirements, i.e., *"what shall be developed"*, from developing the design, i.e., the *"how shall the system be developed"*. In other words, it is important to separate the *problem space* from the *solution space*. The main reason for this separation is to make explicit that there are multiple solutions (hows) for a given problem (what). Specifying directly the solution for a given problem bears the risk to miss other possible solutions that might solve the problem more efficiently. Thus, the selection of solutions for a given problem should be made explicit and based on argumentation rather than on implicit experience.

During the development of a system, there are multiple problem-solution pairs (see for example the metaphor of the Twin Peaks Model [Cleland-Huang et al., 2013; Nuseibeh, 2001]) as they depend on the stakeholder's viewpoints [Davis, 1993; Pohl, 2010]. For example, for a system architect, by specifying a requirement, the requirement engineer defines the problem (what) while the resulting system architecture defines the solutions (hows). In essence, this is a very simplistic view, but nonetheless characterizes the iterations between problems and solutions during development. Ultimately, iterating between problem and solutions in each phase eventually reduces the number of possible solutions and yields to a (hopefully) efficient set of solutions.

2.2. Fundamentals: A Formal System Model

In the remainder of this dissertation, we aim to assess whether a categorization based on a system model is adequate[1] for requirements found in practice. To this end, we introduce in this section the FOCUS theory [Broy, 2010a,b; Broy and Stølen, 2001] and its probabilistic extension [Neubeck, 2012]. The FOCUS theory provides the following characteristics:

- A system is interactive and encapsulates a state that cannot be assessed directly from the outside. A system interacts with its environment exclusively by its interface. An interface is formed by named and typed channels, which are communication links for asynchronous, buffered message exchange.
- A system receives input messages (from its environment) on its input channels and returns output messages (to its environment) over its output channels.
- A system may be underspecified, i.e., nondeterministic. This means that, given a sequence of input messages, there may exist several sequences of output messages that represent the reaction of the system.
- A system may have a probabilistic behavior. This means that, given a sequence of input messages, there may exist several sequences of output messages that may occur with different probability.
- The interaction between a system and its environment takes places concurrently in a global time frame.

In the following, we first introduce basic concepts like numbers, functions, and streams. Then, we introduce nondeterministic interface behaviors and finally probabilistic and nondeterministic interface behaviors. Finally, we introduce the underlying system model.

2.2.1. Functions and Streams

Basic Numbers In the following, we use the notion $\mathbb{N} = \{1, 2, 3, \dots, \}$ for natural numbers excluding zero and $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ for the natural number including zero. We further use the extended natural numbers $\overline{\mathbb{N}_0} = \mathbb{N}_0 \cup \{\infty\}$, the real numbers $\mathbb{R}$, and the extended real numbers $\overline{\mathbb{R}} = \mathbb{R} \cup \{-\infty, +\infty\}$.

Multiset A multiset (or bag) is a generalization of the concept of a set and allows multiple instances of the multiset's elements. Formally, a multiset M over a set A is a mapping from A to $\mathbb{N}_0$. The number $M(x), x \in A$ represents how often the element x appears in the multiset M. We call $\mathbb{N}_0^A$ the set of all multisets over A.

Functions Given a function $f : X \to Y$. We call $dom(f) := X$ the domain and $rng(f) := Y$ the range of the function. For finite sets $X = \{x_1, \dots, x_n\}$, we may define f by

$$f = (x_1 \mapsto f.x_1, \dots, x_n \mapsto f.x_n)$$

Note. *In the following, we use the notion $f.x$ interchangeably with $f(x)$.*

Streams A *stream* is an infinite or finite sequence of elements of a given set. Given a set M of messages, we call the function

$$s : \{0, \dots, n-1\} \to M$$

stream (or sequence) of length n over M. A stream s may be written as

$$s = \langle s.0, \dots, s.(n-1) \rangle$$

An *infinite stream* over the set of messages M is a function $\mathbb{N}_0 \to M$. We write M^n, M^*, M^∞, M^ω for the set of streams of length n, finite streams, infinite streams, and streams of arbitrary length. Due to the functional nature of streams, we can also define the *domain* and *range* of a stream:

$$dom \in M^\omega \to \{[0, \dots, n] \mid n \in \mathbb{N}\} \cup \mathbb{N}$$
$$rng \in M^\omega \to \wp(M)$$

Timed Streams An infinite timed stream represents differs from an ordinary stream in a way that it represents an infinite history of communications over a channel or an infinite history of activities that are carried out sequentially in a discrete time scale[Broy, 2010a]. The discrete time frame represents time as an infinite chain of time intervals of

finite equal duration. In each time interval a finite number of messages is communicated or a finite number of actions is executed.

With

$$(M^*)^\infty$$

we denote the set of timed streams. Elements of $(M^*)^\infty$ are infinite sequences of finite sequences.

Operators on Streams In this work, we will use the following simple operators on streams. For further details on these operators, the reader is referred to [Broy, 2010a,b; Broy and Stølen, 2001].

$\langle\rangle$	Empty sequence or empty stream.
$\langle m \rangle$	One-element stream containing m as its only element.
$x.t$	t-th element of the stream x.
$\#x$	Length of the stream x.
$x^\wedge z$	Concatenation of the stream x with the stream z.
$x \downarrow_t$	Prefix of length t of the stream x.
$C[c/c']$	The renaming of the channel c in the component C to c'
$s\vert_j^i$	Returns the sub stream of s starting from i to j.

Prefix Ordering on Streams We introduce a *prefix ordering* $\sqsubseteq$, which is a partial order on steams. Given streams $x, y \in M^\omega$, the order is defined as

$$x \sqsubseteq y \Leftrightarrow \exists z \in M^\omega : x^\wedge z = y$$

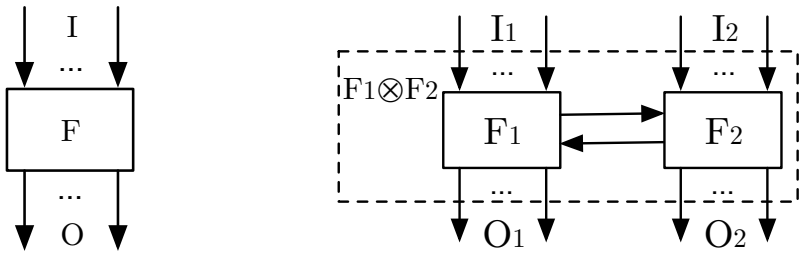
where $x^\wedge z$ denotes the concatenation of sequences, if x is infinite, $x^\wedge z = x$.

2.2.2. Syntactic and Semantic System Interfaces

In Focus, systems are specified by logical expressions relating the input and output histories. In order to compose systems (or components), a composition operator is provided (the so-called parallel composition with feedback). The composition operator allows to model concurrent execution and interaction of systems within a network. To capture the systematic and stepwise development of systems, e.g., from high-level requirements over system requirements to a concrete system implementation, the notion of refinement is introduced. The compositionality of the refinement in Focus guarantees that refinement steps for the system of a composed system realize a refinement step for the composed system, and thus, modularity is guaranteed.

Types A type is a set of data elements. These data elements are then used as messages or as values of state attributes. Let S be a set of types be given. Then,

$$M = \bigcup_{s \in S} s$$



(a) System F with input channels I and output channels O.

(b) Composition of F_1 with F_2.

Figure 2.4.: Graphical representation of a system and of composition of two systems.

is the set of all data messages.

Typed Channels A typed channel is an identifier for a sequential directed communication link for messages of that type. Let C be the set of typed channel names. We assume that a type assignment for the channels is given as follows

$$type : C \to S$$

Channel Valuations Given a set C of typed channels, a channel valuation is an element of the set $\overline{C}$, with

$$\overline{C} = \{x : C \to (M^*)^\infty : \forall c \in C : x.c \in (type(c)^*)^\infty)\}$$

A channel valuation $x \in \overline{C}$ associates a stream of elements of type $type(c)$ with each channel $c \in C$.

Syntactic Interface Let I be a set of typed input channels and O the set of typed output channels. The pair (I, O) characterizes the syntactic interface of a system. Figure 2.4a shows a graphical representation of a system and its interface. The syntactic interface is denoted by

$$(I \blacktriangleright O)$$

Interface Behavior Let I be a set of typed input channels and O the set of typed output channels. We call the function F interface behavior. F is defined as follows

$$F : \overline{I} \to \wp(\overline{O})$$

Given the input history $x \in \overline{I}$, $F.x$ denotes the set of all output histories that a system with behavior F may exhibit on input x. If $F.x$ is a one-element set for every input history $x \in \overline{I}$, we call F deterministic.

Causality in Interface Behaviors Let an interface behavior $F : \overline{I} \to \wp(\overline{O})$ be given. F is called *weakly causal*, if for all $t \in \mathbb{N}, x \in \overline{I}, z \in \overline{I}$, the following holds

$$x \downarrow_t = y \downarrow_t \Rightarrow (F.x) \downarrow_t = (F.z) \downarrow_t$$

Hence, F is weakly causal if the output in the t-th position in the stream does not depend on the input that is received after t. This ensures that there is a proper flow for the system modeled by F. F is called *strongly causal*, if for all $t \in \mathbb{N}, x \in \overline{I}, z \in \overline{I}$, the following holds

$$x \downarrow_t = y \downarrow_t \Rightarrow (F.x) \downarrow_{t+1} = (F.z) \downarrow_{t+1}$$

If F is strongly causal, then, the output in the t-th position does not depend on input that is received after $t - 1$.

Realizability An interface behavior F is called realizable, if there exists a strongly causal total function $f : \overline{I} \to \overline{O}$ such that

$$\forall x \in \overline{I} : f.x \in F.x$$

A strongly causal function $f : \overline{I} \to \overline{O}$ provides a deterministic strategy to calculate for every input history a particular output history that is correct with respect to F.

Full Realizability An interface behavior F is called fully realizable, if it is realizable and if, for all input histories $x \in \overline{I}$

$$F.x = \{f.x : f \in [F]\}$$

holds. Full realizability guarantees that, for all output histories, there is a strategy (a deterministic implementation) that computes this output history.

Composition Given two interface behaviors

$$F_1 : \overline{I_1} \to \wp(\overline{O_1}) \text{ and } F_2 : \overline{I_2} \to \wp(\overline{O_2})$$

with disjoint sets of output channels, i.e., $O_1 \cap O_2 = \emptyset$. Then, we define the parallel composition with feedback by the interface behavior (see Figure 2.4b for a graphical representation of the composition)

$$F_1 \otimes F_2 : \overline{I} \to \wp(\overline{O})$$

where $I = (I_1 \cup I_2) \setminus (O_1 \cup O_2)$ and $O = (O_1 \cup O_2) \setminus (I_1 \cup I_2)$. The resulting function is specified by the following equation (here $y \in C$ where the set of channels C is given by $C = I_1 \cup I_2 \cup O_1 \cup O_2$):

$$(F_1 \otimes F_2).x = \{y|O : y|I = x|I \wedge y|O_1 \in F_1(y|I_1) \wedge y|O_2 \in F_2(y|I_2)\}$$

Refinement (Property Refinement) Given two interface behaviors

$$F_1 : \overline{I} \to \wp(\overline{O}) \text{ and } F_2 : \overline{I} \to \wp(\overline{O})$$

F_1 is refined by F_2, written $F_1 \rightsquigarrow F_2$, if

$$\forall x \in \overline{I} : F_2.x \subseteq F_1.x$$

Refinement allows us to replace an interface behavior with one having additional properties. This way a behavior is replaced by a more restricted one. Obviously, property refinement is a partial order and thus reflexive, asymmetric, and transitive. One can see a refinement step as adding requirements as it is done step by step in requirements engineering.

Compositionality of (Property) Refinement The (property) refinement is compositional in FOCUS. Given two pairs of interface behaviors

$$F_1 : \overline{I_1} \to \wp(\overline{O_1}) \text{ and } F_2 : \overline{I_2} \to \wp(\overline{O_2})$$

and

$$F_1' : \overline{I_1} \to \wp(\overline{O_1}) \text{ and } F_2' : \overline{I_2} \to \wp(\overline{O_2})$$

then, the compositionality of refinement is expressed by the following rule:

$$\frac{F_1 \rightsquigarrow F_1' \quad F_2 \rightsquigarrow F_2'}{F_1 \otimes F_2 \rightsquigarrow F_1' \otimes F_2'}$$

2.2.3. Probabilistic System Behavior

In this section, we briefly introduce basic concepts of probability theory. We largely follow the presentation of Neubeck [2012] and Junker [2016].

σ-field Given a set Ω, a set $\mathcal{F} \subseteq \wp(\Omega)$ is called a σ-field, if $\mathcal{F}$ is closed under complements and countable unions and contains $\emptyset$. Due to its closure properties, a σ-field also always contains Ω. The pair $(\Omega, \mathcal{F})$ is called a *measurable space* and the elements of $\mathcal{F}$ are called *measurable sets*.

If $\mathcal{G}$ is a subset of $\wp(\Omega)$, then with $\sigma(\mathcal{G})$ we denote the σ-field generated by $\mathcal{G}$, that is, the smallest σ-field that subsumes $\mathcal{G}$, which is guaranteed to exist. As elaborated by Neubeck [2012], one can construct σ-fields $\mathcal{S}_n$ over finite streams of length n and a

σ-field $\mathcal{S}$ over infinite streams. In particular, $\mathcal{S}$ is the σ-field generated by the set of *basic cylinders*, i.e., a set of the form $\mathcal{C}(x) = \{y \in \Sigma^\infty : x \sqsubseteq y\}$, for $x \in \Sigma^*$.

Probability Space A probability space is a triple $(\Omega, \mathcal{F}, \mu)$ where $(\Omega, \mathcal{F})$ is a measurable space, μ is a function $\mathcal{F} \to \mathbb{R}$ such that

- $\forall A \in \mathcal{F} : \mu(A) \geq 0$
- $\mu(\bigcup_{i=1}^{\infty} A_i) = \Sigma_{i=1}^{\infty} \mu(A_i)$ for pairwise disjoint sets $(A_k)_k$
- $\mu(\Omega) = 1$

Probability Space for Infinite Streams As shown by Neubeck [2012], a probability space for infinite streams $(\Sigma^\infty, \mathcal{S}, \mu)$ can be derived from consistent probability spaces $(\Sigma^n, \mathcal{S}_n, \mu_n)$ for finite streams. Then, μ is the probability measure on infinite streams that is consistent with all probability measures μ_n for fixed prefixes.

In the following, we will denote the set of possible probability measures for a set of streams Σ^∞ with $\mathbf{Pr}(\Sigma^\infty)$. We will furthermore use the notation $\mathbf{Pr}[q(\mu)]$ as short form for $\mu(\{\omega\}q(\omega))$ for a predicate on streams q.

Probabilistic Interface Behavior Let I be a set of typed input channels and O the set of typed output channels. We call the function F probabilistic interface behavior. F is defined as follows

$$F : \overline{I} \to \wp(\mathbf{Pr}(\overline{O}))$$

F maps input histories to sets of probability measures over output histories. For a more detailed discussion on the theoretical basis of the probabilistic extension of FOCUS, the reader is referred to the work of Neubeck [2012] and Junker [2016].

Note. *Neubeck [2012] further provides notions of causality, composition, compositionality, etc. that extend the previously introduced notions to the probabilistic case.*

2.2.4. System, Components, and Functions

Figure 2.5 gives an overview of our understanding of a system, its components, and its functions: A system interacts over a set of typed input/output channels with its environment. It is composed of a set of components (C_1, C_2, and C_3 in the figure), which communicate over a set of typed input output channels with each other. Functions provide a different view on the system; The overall functionality of a system is decomposed according to features that are provided to a user. The purpose of a system is to offer a set of user functions that serve a specific purpose [Broy et al., 2007]. Thus, a system provides a set of functions (F_1 and F_2 in the figure) that communicate via typed input/output channels with its environment.

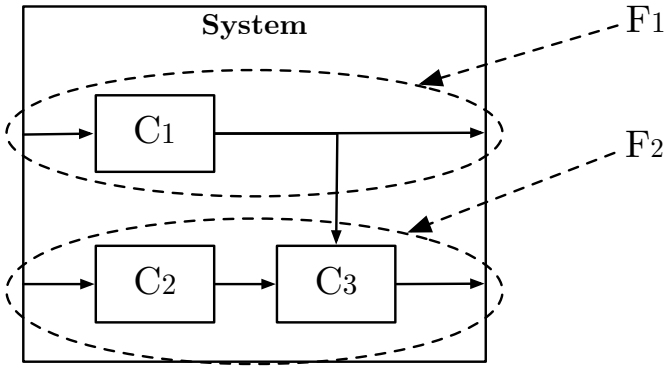


Figure 2.5.: System. Component, and Functions.

System More formally, a system has a unique identifier s and is specified by a syntactic interface $(I \blacktriangleright O)$ and probabilistic interface behavior

$$S : \overline{I} \to \wp(\mathbf{Pr}(\overline{O}))$$

Behavior Function For reasons of simplicity, we introduce the behavior function $\mathcal{B}$, that returns for a given identifier the respective probabilistic interface behavior. Thus, given a function, component, or system with identifier id and syntactic interface $(I \blacktriangleright O)$ and probabilistic interface behavior $F : \overline{I} \to \wp(\mathbf{Pr}(\overline{O}))$, the following holds:

$$\mathcal{B}(id) = F$$

Component A component has a unique identifier c_i and is specified by a syntactic interface $(I \blacktriangleright O)$ and probabilistic interface behavior

$$C : \overline{I} \to \wp(\mathbf{Pr}(\overline{O}))$$

Given a system with identifier s and syntactic interface $(I_s \blacktriangleright O_s)$ and a set of components with identifiers $c_1, \ldots, c_n$ and syntactic interface $(I_{c_1} \blacktriangleright O_{c_1}), \ldots, (I_{c_n} \blacktriangleright O_{c_n})$, then the composition of the behaviors of all c_i is equal to the behavior of the system.

$$\mathcal{B}(s) = \bigotimes_{i \in \{1,\ldots,n\}} \mathcal{B}(c_i)$$

Function A function has a unique identifier f_i and is specified by a syntactic interface $(I \blacktriangleright O)$ and probabilistic interface behavior

$$F : \overline{I} \to \wp(\mathbf{Pr}(\overline{O}))$$

A function describes the structure and the behavior of the system from a black-box perspective (i.e. at the interface of the system). Thus, given a system with identifier s

and syntactic interface $(I_s \blacktriangleright O_s)$, then $I \subseteq I_s$ and $O \subseteq O_s$. Furthermore, given a set of functions with identifiers $f_1, \ldots, f_n$ and syntactic interface $(I_{f_1} \blacktriangleright O_{f_1}), \ldots, (I_{f_n} \blacktriangleright O_{f_n})$, then the composition of the behaviors of all f_i is equal to the behavior of the system.

$$\mathcal{B}(s) = \bigotimes_{i \in \{1,\ldots,n\}} \mathcal{B}(f_i)$$

Requirements In terms of the system modeling theory, a requirement describes desired observations of behavior between a system and its environment. Therefore, a requirement for a probabilistic system is expressed by a predicate over histories of input streams and probability distributions over output histories:

$$R : \overline{I} \times \mathbf{Pr}(\overline{O}) \to \mathbb{B}$$

A system, component, or function with identifier id fulfills a requirement R with the same syntactic interface, iff

$$R \rightsquigarrow \mathcal{B}$$

In the following, we discuss the scope of a requirement; It may restrict the behavior of an individual system, but also restrict the behavior of a set of systems. The former case is already well understood by e.g. Broy [Broy, 2010a,b; Broy and Stølen, 2001] in the timed and untimed, deterministic and nondeterministic case and by Neubeck [2012] in the probabilistic case. For the latter, i.e., requirements that restrict the behavior of a set of systems, we discuss possible application scenarios for the probabilistic case and show how one can construct a probability space over a (finite) set of systems by a product space construction (see e.g. [Bauer, 2001; Gray et al., 2001; Pollard, 2002]).

Requirements over Sets of Systems

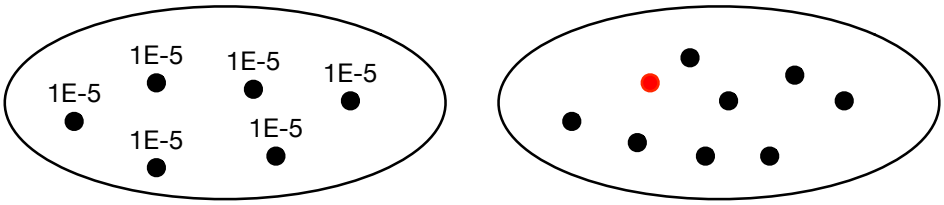
Usually, the scope of a requirement is a system, i.e., a requirement describes a desired property of a system. For example, consider the following (probabilistic) requirement:

"The probability that the airbag of the system fails to work in case of a crash shall be less than 0.01%."

Given such a requirement, we can for example check whether a system at hand fulfills that requirement. This requirement may describe a deterministic behavior, it may describe a nondeterministic behavior, it may describe a (deterministic) probabilistic behavior, or it may describe a (nondeterministic) probabilistic behavior. All these cases are already well understood by, e.g., Broy [Broy, 2010a,b; Broy and Stølen, 2001] (in the deterministic and nondeterministic case) and Neubeck [2012] (in the probabilistic case).

However, there are some cases where we do not want to restrict the behavior of a single system, but the behavior of a set of systems, e.g., the set of all vehicles of a car manufacturer. For example, we may want to formulate a (probabilistic) requirement like

"The probability that no airbag of a vehicle ***out of the set of all our vehicles*** *fails to work in case of a crash shall be more than 0.99%."*



(a) A set of systems with probabilistic behavior.

(b) A set of systems with one broken system (indicated as red circle).

Figure 2.6.: Two different interpretations of probabilistic requirements over sets of systems.

We can understand this requirement in two different ways (see Figure 2.6):

1. Consider a probabilistic description of a the airbag with a probability to fail of 0.001% and 10 identical (and independent) systems (see Figure 2.6a). Then, we can compute the probability that no airbag fails as the product of the individual probabilities, as the individual systems are independent of each other.

$$\begin{aligned}\mathbf{Pr}(\forall s \in S\colon \neg \mathit{Fail}(s)) &= \Pi_{s\in S}\mathbf{Pr}(\neg \mathit{Fail}(s)) \\ &= \Pi_{s\in S}(1 - \mathbf{Pr}(\mathit{Fail}(s))) \\ &= (1 - 0.00001)^{10} \\ &= 0.9999000045\ (> 0.99\%)\end{aligned}$$

   Thus, in our example, the requirement holds. However, a probabilistic description of a system is just a means to model uncertainty about the behavior of the system (similar to nondeterministic or fuzzy descriptions [Koutsoumpas, 2015]). Thus, if we pick a real system, the airbag *either works or does not work* (if we only consider deterministic systems). We discuss this case in the following.

2. Following this argument, if we consider only deterministic systems, the airbag of a given system either works or does not work. It may be broken because of some failures in the production process. In this case, we can also understand the requirement as

   "The probability that the airbag of one randomly chosen vehicle out of the set of all our vehicles fails to work in case of a crash shall be less than 0.01%."

   In this case, we can leverage the concept of *a priori probabilities* (see, e.g., [Mood et al., 1974]), i.e., the probability that is derived purely by deductive reasoning. One way of deriving a priori probabilities is the principle of indifference, i.e., if there are N mutually exclusive and exhaustive events and if they are equally likely, then the probability of a given event occurring is $\frac{1}{N}$.

   Let us consider an example: Given a set of systems of size 20000. We know that because of the production process, there is one system out of 20000, with a broken

airbag. If we know that the airbag of one of our 20000 systems is broken, we can derive the *a priori probability* as follows (as we have mutually exclusive and exhaustive events):

$$\mathbf{Pr}(\exists s \in S\colon \mathit{Fail}(s)) = \frac{1}{|S|} = \frac{1}{20000} = 0.00005\ (< 0.01\%)$$

Thus, again, the requirement holds. However, in this case, the probability only depends on the number of broken systems in relation to the total number of systems.

Practical Relevance of Requirements over Sets of Systems Both cases are practically relevant. As already discussed in our example, for a car manufacturer, it is highly important to calculate the defect probability of the set of their systems by probabilistic models (case 1). This may be due to legal reasons. This case is also relevant for safety and availability requirements. For example, there may be a probabilistic description of a system which includes the description of random failures (see e.g. [Börcsök, 2011]). Then, we may want to formulate requirements that describe the absence of those failures with a certain probability for one system. If there are multiple systems shipped, it is also of importance to formulate requirements over this set of systems.

The second case is also practically important. For example, a production process may introduce systematic failures (see e.g. [Börcsök, 2011]) for certain subsystems. Then, if multiple of those subsystems are integrated, we may be interested in the probability that a given system contains a broken subsystem.

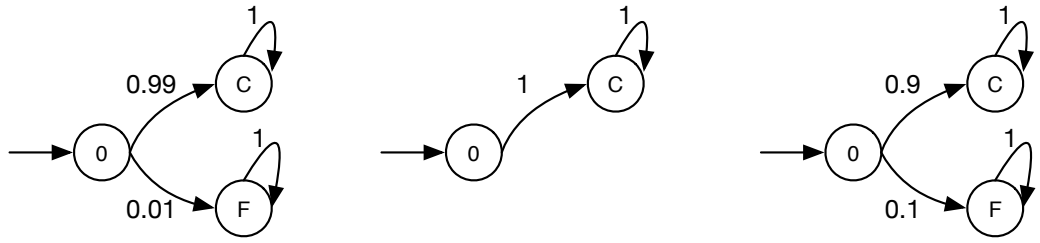
In the following, we will focus on case 1: requirements that describe a property over a set of systems with probabilistic behavior. In particular, we provide a formal notion of a requirement over such a set of systems.

Requirements over Sets of Systems with Probabilistic Behavior

In order to introduce probabilistic behavior for a single system, we need to construct a probability space $(\Omega, \mathcal{B}, \mu)$. However, as we have infinite streams as our sample space Ω (e.g. for a given set of typed channels O, we have $\Omega = \overline{O}$), we face the challenge to create a measure on subsets of Ω, which is uncountable. The work of Neubeck [2012] shows how to construct a measurable space and a probability measure μ_i for Ω[5]. They extend the concept of a product space $(\Sigma^\infty, \sigma(\wp(\Sigma)^\infty))$ as introduced by Gray et al. [2001] by cylinders instead of rectangles, i.e., sets of the form $\{\rho \in (S)^\infty \mid \varphi \sqsubseteq \rho\}$. Then, they construct a unique probability measure μ_i on the product space by the extension theorem of Carathéodory. This enables to formulate requirements as predicates over histories of input streams and probability distributions over output histories: $R \subseteq \overline{I} \times \mathbf{Pr}(\overline{O})$.

If the scope of a requirement is a set of systems, we can leverage the construction by Neubeck. However, in contrast to Neubeck's construction, we want to formulate requirements over a set of systems S (e.g., the set of all vehicles of a specific company).

[5]Note that μ_i depends on the input i.



(a) Probabilistic behavior of S_1. (b) Probabilistic behavior of S_2. (c) Probabilistic behavior of S_3.

Figure 2.7.: Three example systems S_1, S_2 and S_3. The state F indicates failure. The systems are independent of the input for reasons of simplicity. Thus, the probability measure is also independent of the input.

In order to formulate those requirements, we need to construct a probability space over sets of systems. We construct the measurable product space as follows:

Given an arbitrary set S of (probabilistic) systems, each over typed input channels I and typed output channels O, with their associated σ-fields $A_s = (\Omega_s, \mathcal{A}_s)$ and probability measure $\mathbf{Pr}_{s,i}$ for $s \in S$ and $\Omega = \overline{O}$ (as introduced by Neubeck [2012]). Note that the probability measure $\mathbf{Pr}_{s,i}$ depends on the input $i \in \overline{I}$ for each system $s \in S$. Then, we can construct the product space $\Pi_{s \in S} A_s$ as follows:

$$(\Pi_{s\in S}\Omega_s, \underbrace{\{\{\omega \in \Pi_{s\in S}\Omega_s \mid \forall j \in J, \omega_j \in M_j\} \mid J \subseteq S \text{ finite}, \forall j \in J\colon M_j \in \mathcal{A}_j\}}_{\text{The set of all finite cylinders over } \Pi_{s\in S}\Omega_s})$$

Note (Independence Assumption)**.** *Note that we base this construction on one basic assumption: The individual systems in S are assumed to be independent of each other. This assumption allows us to construct the product space. If the systems are not independent of each other, we have to construct one system which is composed of multiple subsystems. This may for example be needed if we consider systems whose failure probability depends on the number of created systems or if the systems interact with each other.*

Then, according to Bauer [2001] and Pollard [2002], we know that there exists a unique probability measure $\mathbf{Pr}_{\Pi,i}$ over $\Pi_{s\in S} A_s$ with the property that for finite $J \subseteq S$ and $M_j \in \mathcal{A}_j$,

$$\mathbf{Pr}_{\Pi,i}(\forall j \in J\colon X_j \in M_j) = \Pi_{j\in J}\mathbf{Pr}_{j,i}(M_j)$$

Furthermore, given that the set of systems S is finite, we can directly calculate this probability. Thus, for reasons of simplicity, we require the set of systems to be finite for the remainder of this thesis. Now, given a finite set of systems S, and a a family of properties P_s, we can formulate requirements like

- *"The probability that all systems s out of S fulfills property P_s shall be greater than 0.999"*, which is

$$\mathbf{Pr}_{\Pi,i}(\forall s \in S\colon P_s) = \Pi_{s \in S}\mathbf{Pr}_{s,i}(P_s)$$

Thus, the probability that P_s is fulfilled by every system $s \in S$ is the product of the individual probabilities. For example, let us consider the three systems in Figure 2.7. In these systems, the state F indicates failure and the state C indicates correct behavior. Thus, in system S_1 the probability that the system works correct is 0.99, in system S_2 it is 1, and in system S_3 it is 0.9. Now we can formulate the requirement

"The probability that all systems s out of $\{S_1, S_2, S_3\}$ work correct shall be greater than 0.9".

Then, we can calculate this probability by

$$\begin{aligned}\mathbf{Pr}_{\Pi}(\forall s \in S\colon P_s) &= \Pi_{s \in S}\mathbf{Pr}_{s}(P_s)\\ &= \mathbf{Pr}_{S_1}(P_{S_1}) \cdot \mathbf{Pr}_{S_2}(P_{S_2}) \cdot \mathbf{Pr}_{S_3}(P_{S_3})\\ &= 0.99 \cdot 1 \cdot 0.9\\ &= 0.891\end{aligned}$$

Thus, the requirement is not fulfilled by the set of systems $S = \{S_1, S_2, S_3\}$.

- *"The probability that one system s out of S does not fulfill property P_s shall be less than 0.0001"*, which is

$$\begin{aligned}\mathbf{Pr}_{\Pi,i}(\exists s \in S\colon \neg P_s) &= 1 - \mathbf{Pr}_{\Pi,i}(\forall s \in S\colon P_s)\\ &= 1 - \Pi_{s \in S}\mathbf{Pr}_{s,i}(P_s)\\ &= 1 - \Pi_{s \in S}\left(1 - \mathbf{Pr}_{s,i}(\neg P_s)\right)\end{aligned}$$

Again, consider, for example, the three systems in Figure 2.7. In these systems, the state F indicates failure and the state C indicates correct behavior. Thus, in system S_1 the probability that the system works incorrect is 0.01, in system S_2 it is 0, and in system S_3 it is 0.1. Now we can formulate the requirement

"The probability that one system s out of $\{S_1, S_2, S_3\}$ works incorrect shall be less than 0.01".

Then, we can calculate this probability by

$$\begin{aligned}
\mathbf{Pr}_{\Pi}(\exists s \in S\colon \neg P_s) &= 1 - \mathbf{Pr}_{\Pi}(\forall s \in S\colon P_s) \\
&= 1 - \Pi_{s \in S} \mathbf{Pr}_s(P_s) \\
&= 1 - \Pi_{s \in S}(1 - \mathbf{Pr}_s(\neg P_s)) \\
&= 1 - ((1 - \mathbf{Pr}_{S_1}(\neg P_{S_1})) \cdot (1 - \mathbf{Pr}_{S_2}(\neg P_{S_2})) \cdot (1 - \mathbf{Pr}_{S_3}(\neg P_{S_3})) \\
&= 1 - ((1 - 0.01) \cdot (1 - 0) \cdot (1 - 0.1)) \\
&= 1 - 0.891 = 0.109
\end{aligned}$$

Thus, the requirement is not fulfilled by the set of systems $S = \{S_1, S_2, S_3\}$.

2.3. Requirements Categorization based on a Formal System Model

In this dissertation, we assess whether a categorization based on a system model is adequate[1] for requirements found in practice and whether it effectively supports subsequent development activities. To this end, we introduce in this section an approach for categorizing requirements based on the system model provided by the FOCUS theory [Broy, 2015, 2016]. We follow largely the presentation of Broy [2016].

Broy introduces an approach for categorizing system and software requirements with the goal to provide a sufficiently precise notion and models of adequate systems. He argues that current approaches that categorize requirements into "non-functional" and "functional" are simplistic and not very helpful for classification. Furthermore, he argues that the challenge to characterize classes of functional and non-functional requirements depends on characterizations of the different kinds of observations made about systems captured and formalized in terms of adequate system models. To make these categories precise, he suggests to use structured views onto systems. In particular, he differentiates a number of viewpoints and levels of abstraction. One level addresses a functional view that is adequate to model functional requirements. Other level address more implementation-oriented, technical views onto systems, which then might help to address implementation-specific properties.

In essence, he distinguishes between *behavioral properties* and *representational properties*. Behavioral properties subsume traditional functional requirements, such as *"the user must be able to remove articles from the shopping basket"* as well QRs that describe behavior such as *"the system must react on every input within 10ms"*. Representational properties include QRs that determine how a system shall be syntactically or technically represented, such as *"the software must be implemented in the programming language Java"* [Broy, 2015, 2016]. In the following sections, we introduce his categorization.

2.3.1. Categorization based on a System Model

For categorizing requirements, Broy follows a system-structuring and modeling-oriented approach. In particular, he uses the two following modeling frameworks:

		Syntactic Basis	Behavior	
			Logical	Probabilistic
Black Box View	**Functional View**	Syntactic interface	Logical interface behavior	Probabilistic interface behavior
Glass Box View	**Architecture View**	Hierarchical data flow graph of subsystems	Subsystems and their logical interface behavior	Subsystems and their probabilistic interface behavior
	State View	State space structure (attributes) I/O messages	Logical state machine Logical state transitions	Probabilistic state machine Probabilistic state transitions

Figure 2.8.: Overview of Behavioral Categories (Adapted from Broy [2016]).

- A system notion and a system modeling theory, supporting several fundamental system views such as the *interface view*, the *architecture view*, and the *state view*. For these views, he distinguishes between (i) a *logical view* in terms of sets of observations and sets of instances of behavior and (ii) a *probabilistic view* in terms of probabilities of observations and instances of behaviors.

- A comprehensive system architecture, comprising a context view, a functional view, an architectural sub-system view, and a technical view including aspects of software, hardware, and mechanics as well as physical realizations.

2.3.2. Behavioral Properties: System Behavior

Behavioral views address the behavior over the interface of a system as well as the internal behavior[6]. There are two principal means to describe the internal behavior of systems: state machines and architectures, both describing the system's internal structure, state space, state transitions, the structuring into its sub-systems, their connection, relationship, and interaction. The state view is described by the states, given by the state space, which the system can take, the state transitions, and the probability that certain state transitions are taken. Figure 2.8 shows the proposed categorization for behavioral properties.

Functional Properties: Logical and Probabilistic Interface Behavior

In the functional view, the functionality of the system is specified by means of the interface behavior of the system. It is described by the interface behavior of the system and may include both logical and probabilistic views. As a result, we understand how the system cooperates with its environment exclusively considering issues of interactions. In both these views we get the information, which interactions are possible in principle and what their probabilities are.

[6]Internal behavior in terms of architecture and state transitions which is hidden from the interface view by information hiding.

Behavioral Glass Box View: Logical and Probabilistic Behavior

The glass-box view addresses the internal structure and properties of systems. The glass-box view consists of two complementary views: the architectural view and the state view. They are complementary in the sense, that they describe a system's architecture in terms of its set of sub-systems with their behavior and interaction and the system's state space and state transitions.

- **Architectural View: Structure, Logical and Probabilistic Behavior:** In the architectural view, a system is decomposed into a hierarchy of sub-systems forming its components [Broy, 2010b]. The architecture describes how the components are connected and their behavior is described by the interface behaviors of the components in terms of their interactions. By the description of the behaviors of its components and the structure of the architecture the behavior of the architecture is specified from which the interface behavior of the system can be deduced. In the architectural view, we get a view onto the behavior of a system in terms of its sub-systems.

- **State View: Logical and Probabilistic State Transitions:** In a state view a system is described by a state machine [Vogelsang, 2015; Vogelsang et al., 2015]. The state space of the system is described and the state transition relation is specified including inputs and outputs. This yields a (probabilistic) Mealy machine extended to possibly infinite state space. The machine represents the logical state view.

2.3.3. Non-behavioral Properties: System Representation

Besides behavioral properties, there are non-behavioral properties of systems, including properties that refer to the syntactic representation of systems. For example, quality attributes such as the readability of code. A rich class of such properties is found in the technical views onto systems; for instance, this covers properties such as material or geometry.

In the behavioral categories, we describe by which modeling concept a system part is represented. For software, we may require the representation in a specific programming language or a specific coding standard, as for example the requirement "the software must be implemented in the programming language Java". The technical view refers to a large extent to the question how a system is represented. Therefore, certain technical standards or specific devices might be suggested or even strictly required. The system model leads to a clear separation and categorization of properties into the following two categories of system properties:

Behavioral properties including probabilistic and logical behavior addressing interface, architectural, state transition, and technical views onto systems.

Representation related properties that describe the way the system is syntactically or technically represented, described, structured, implemented, and executed (such as which programming languages are used etc.).

2.3.4. Summary of Broy's Categorization

In summary, Broy provides a systematic requirements categorization according to three fundamental views:

Black-box Interface View In this view, the functionality of the system is specified by means of the interface and interface behavior in terms of the interaction over the system boundaries.

(Logical) Sub-system Structuring View This view contains logical and probabilistic properties at the level of architecture—including state views of state spaces and state transitions. This view is captured by architecture and architectural behavior or by state and state-transition behavior.

Representation View This view includes technical, physical, syntactical representation, structuring, and implementation details.

For behavior, Broy further distinguishes between logical behavior in terms of sets of patterns of interaction and probabilistic behavior in terms of the probabilities for the patterns of interaction.

To demonstrate the idea, Broy further provides an estimation about the intensity of the relationship between a set of quality attributes and the proposed requirements categorization. He uses a set of quality attributes from Lochmann and Wagner [2012]. Figure 2.9 depicts this relationship; The rows represent the quality attributes and the columns the different views, separated in syntactic, logical, and probabilistic, where applicable. The intensity of the relationship is indicated by the color of the cell: from black (very strong) to white (none).

The table indicates that certain system properties are not referring to behavior directly but to aspects of representation, e.g., readability. Furthermore, there is a rich class of requirements that are mixtures of properties from several categories. According to his estimation, only a restricted set of the QRs is related to non-behavioral issues when using the proposed categorization.

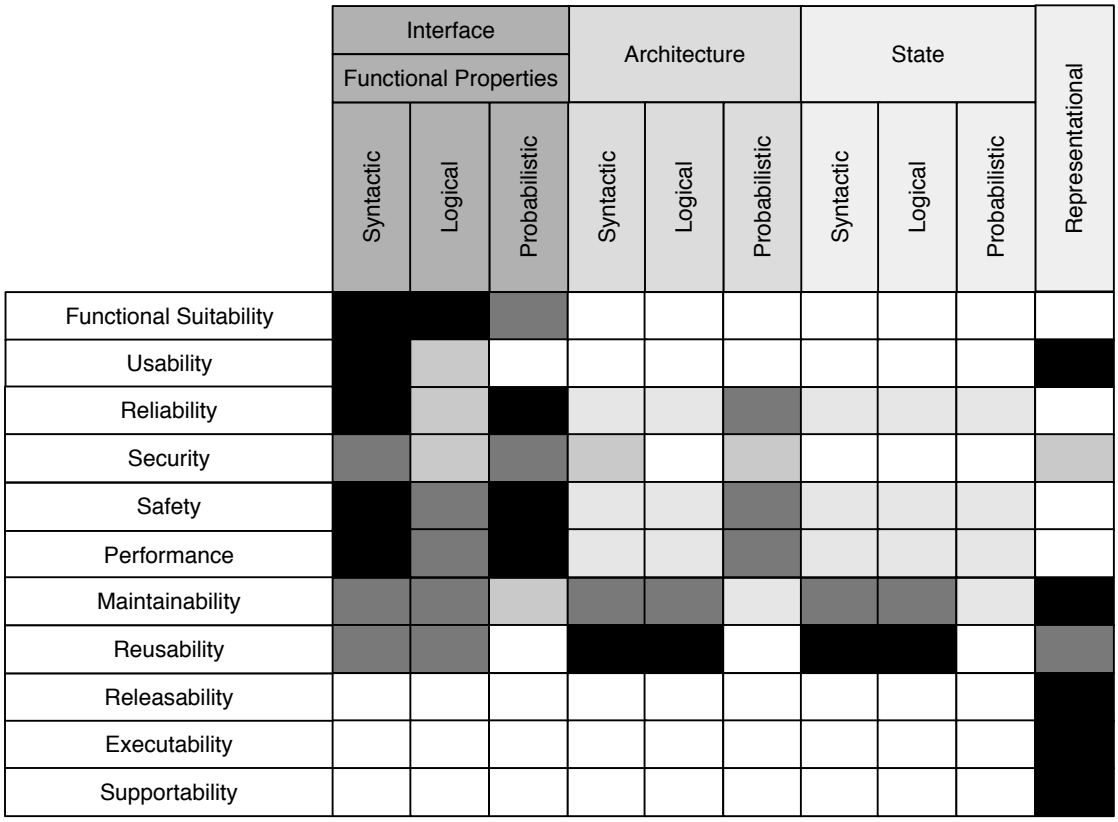


Figure 2.9.: Intensity of the relationship between quality attributes and modeling views. The intensity of the relationship is depicted from black (very strong) to white (none) (Adapted from Broy [2016]).

"We are social creatures to the inmost center of our being. The notion that one can begin anything at all from scratch, free from the past, or unindebted to others, could not conceivably be more wrong."

— Karl Popper

Chapter 3
Related Work

To set the scope of this dissertation, we discuss work related to requirements categorizations, including a historical sequel. Furthermore, to get an understanding of how QRs are handled in practice, we further discuss empirical studies on QRs that analyze how practitioners deal with QRs. Related work concerning the specific topics of the chapters will be discussed in detail in each of the respective chapters.

3.1. Requirement Categorizations

This dissertation is about requirements categorizations. Thus, in this section, we first provide a historical sequel to get an understanding on the reasons why there is a traditional distinction between functional and quality requirements. Then, to get an overview on requirements categorizations, we describe relevant categorizations and finally provide a critical discussion on categorizations.

3.1.1. Historical Note on Requirements Categorizations

In the early days of computer science, research was mostly concerned with handling very sparse computing and memory resources. The focus was rather on efficiently building computing machines than on designing programs. At that time, Zuse built the Z1 (in 1938), the first freely programmable computer which used Boolean logic and binary floating point numbers [Rojas, 1997]. Slowly, but steadily, the focus shifted from hardware design to also recognizing programming as a great challenge. In the early-1960s, first assembly languages were invented and subsequently, in the mid-1960, first universal

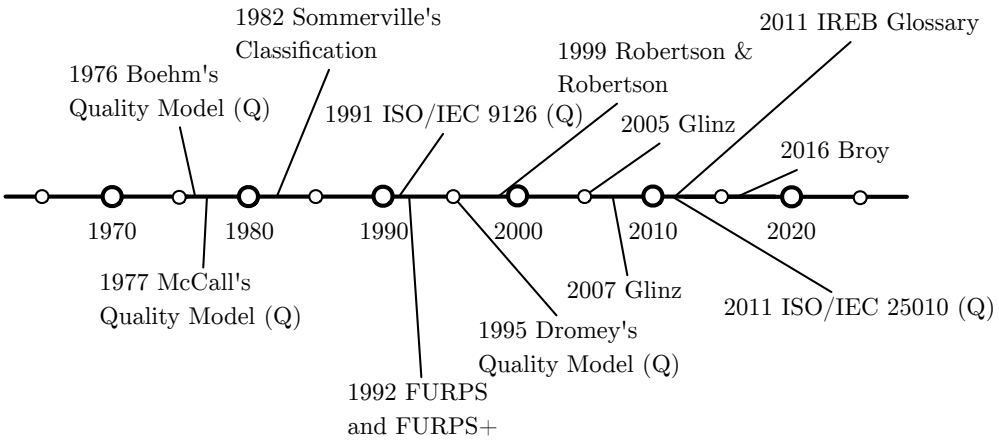


Figure 3.1.: Historical sequel of quality models and requirement categorizations considered in this work. Quality models are marked with (Q).

programming languages, as for example PL/1, ALGOL-W, and Simula 67. Still, the challenge was to write programs that produced the desired results.

As the computing and memory resources increased, more complex software projects were possible at the cost of greater programming effort, higher amounts of source code, and an increasing number of project failures. In these days, Floyd [1967], Hoare [1969], McCarthy [1961], and Dijkstra [1959], just to name a few, worked on the semantics of programs and verification, i.e., functional correctness. In the course of the software crisis in the mid-1960, research started to acknowledge that programming alone is not enough and that an engineering approach is needed. To put it in the words of Dijkstra:

> *"The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem."*
>
> — Edsger W. Dijkstra – The Humble Programmer [Dijkstra, 1972]

Consequently, the term *software engineering* [Bauer et al., 1968] was coined in 1968 in Garmisch-Patenkirchen. The name *software engineering* was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering [Bauer et al., 1968]. In this context, Floyd, Hoare, and Dijkstra introduced *structured programming* [Dahl et al., 1972], which broadened the goal of programming from the mere coding of an algorithm to improving the clarity, quality, and development time of programs. For example, Dijkstra [1968] published in 1968 his famous article "Go to Statement Considered Harmful" with the main argument that `go to` statements have a negative impact on program comprehension. Some years

later, Parnas [1972] published his work "On the criteria to be used in decomposing systems into modules" in 1972 with the goal to improve the flexibility and comprehensibility of a system. Today, we would subsume both of these works under the quality *maintainability*. The tremendous rise of computing resources led to more and more aspects that had to be considered for building high quality systems. Finally, Boehm presented in 1976 the first quality model that tries to capture these aspects.

In summary, when computing resources were sparse, the challenge was to get the functionality of a system right (i.e., implementing *functional requirements*). The focus shifted to other aspects of a system (i.e., implementing *quality* or *non-functional* requirements) with the introduction of structured programming and abstraction mechanisms. We argue that the traditional distinction between *functional* and *quality* requirements has its roots in this development.

3.1.2. Overview of Requirement Categorizations

Figure 3.1 shows a historical sequel of the quality models and requirements categorizations considered in this work. There are three fundamentally different approaches for categorizing requirements:

- **Categorization based on a Quality Model:** These categorizations rely on the close interconnection between quality and requirements. According to the ISO/IEC 9000-2000 [2000], quality is the "degree to which a set of inherent characteristics fulfills requirements" and a requirement is a "need or expectation that is stated, generally implied or obligatory".

  There is much work on quality models in literature. One of the earliest quality models was provided by Boehm in 1976 [Boehm, 1976]. Several other quality models were created based on or in parallel with Boehm's quality model, including McCall's quality model [McCall et al., 1977], the ISO/IEC 9126-2001 [2001], Dromey's Quality Model [Dromey, 1995], and the ISO/IEC 25010-2011 [2011]. All these software/system quality models describe quality characteristics or attributes that categorize aspects of the overall product quality. Then, requirements are categorized according to the quality characteristics of the quality model.

- **Categorization based on Reasoning:** In contrast to categorizations based on a quality model, categorizations based on reasoning are intended to directly categorize requirements into different classes, like for example into *functional requirements* and *non-functional requirements*.

  The first requirements categorization based on reasoning was provided by Sommerville in 1982 [Sommerville and Sawyer, 1997]. After this, several other requirements categorizations were proposed, including the FURPS and FURPS+ categorization [Grady, 1992], the categorization by Robertson & Robertson [Robertson and Robertson, 2012], and the two categorizations by Glinz [Glinz, 2005, 2007].

- **Categorization based on a System Model:** These categorizations classify requirements based on a structured system reference model. So far, the first categorization based on a system model was introduced by Broy in 2016 [Broy, 2016].

In the following, we will provide a short overview of the quality models and the categorizations. This description is based on the work of Mager [2015].

Boehm's Quality Model Boehm's quality model [Boehm, 1976] divides the overall product quality into three characteristics: portability, usability (*as is utility*), and maintainability. These high-level characteristics answer the main questions of the person who buys software:

- Can the product be used in a different environment? (*Portability*)
- How well can the software product be used as-is? (*Usability*)
- How well can the software product be understood, changed and retested? (*Maintainability*) [Al-Qutaish, 2010]

Boehm defines six mid-level characteristics, that represent the expected qualities of the software system: reliability, efficiency, human engineering, testability, understandability and modifiability. These mid-level characteristics further consist of primitive characteristics that enable the creation of quality metrics, for measuring the fulfillment of the qualities.

McCall's Quality Model McCall's quality model [McCall et al., 1977], was developed for the US military to improve the understanding between users and developers. It consists of three different perspectives for defining and identifying the needed qualities:

- **Product revision**: Quality characteristics related to maintaining the product in its current environment.
- **Product transition**: Quality characteristics related to porting the product to a new environment.
- **Product operations**: Quality characteristics related to using the product.

Each of the perspectives contains a set of factors, which are composed of a set of criteria. For each criterion, McCall proposes one or more metrics for measuring the fulfillment. Metrics can be used to measure not only quality criteria, but also quality factors. For example, mean time to failure (MTTF) can be used to measure reliability. McCall's model uses a weighting system to calculate the degree to which a quality factor is present. For this, each quality criterion is assigned a relative importance regarding to the quality factor.

Sommerville's Classification Sommerville categorizes requirements into "functional", "non-functional", and "domain". "Non-functional" requirements are further structured in three general groups: process, product and external considerations. These considerations can be mapped to three different requirement types:

- **Organizational requirements**: Developed from internal policies and procedures in the organization.
- **Product requirements**: Define the product behavior like performance, reliability or portability.
- **External requirements**: Comprise requirements from external factors like interoperability, legislative or ethical requirements.

Each of these considerations contain multiple types and sub-types. In total, it contains 14 NFR types. He requires that that all requirements should be objectively testable and some example objective measures for speed, size, etc. are given. Domain requirements can be either functional or non-functional, as they are new functional requirements, constraint existing requirements or define how specific calculations are done.

ISO/IEC 9126-2001 [2001] The ISO/IEC 9126-2001 [2001] is a standard for quality assurance. The standard defines a model with a multilevel hierarchy of different quality models that contain characteristics and possibly sub-characteristics. Each sub-characteristic is decomposed into attributes that have to be defined by the requirements engineer and must be measurable to verify the quality of the software product.

The standard distinguishes between *internal quality*, *external quality*, *quality in use* and *process quality*.

- *Internal quality* is the quality of the system that can be measured with white box tests. Internal quality attributes can be measured during any stage of development using source code, specifications and other available documents.
- *External quality* is the quality of the system that can be measured with black box tests. External quality attributes can be measured by testing, operating and observing the executable software or system.
- *Quality in use* is the quality of the system for the user. Quality in use can be measured by the user's view in specific user-task scenarios.
- *Process quality* is the quality of the product development process.

As shown in Figure 3.2, the quality model for internal and external quality is further refined into the sub-categories *functionality*, *reliability*, *usability*, *efficiency*, *maintainability*, *portability*. Table 3.1 gives a short description of the high-level quality characteristics.

FURPS and FURPS+ FURPS [Grady, 1992] is a classification model for requirements and the acronym represents the following factors:

- ***F**unctionality*: Functions it performs, their generality and security.
- ***U**sability*: Aesthetics, consistency, documentation.
- ***R**eliability*: Frequency and severity of failure, accuracy of output.

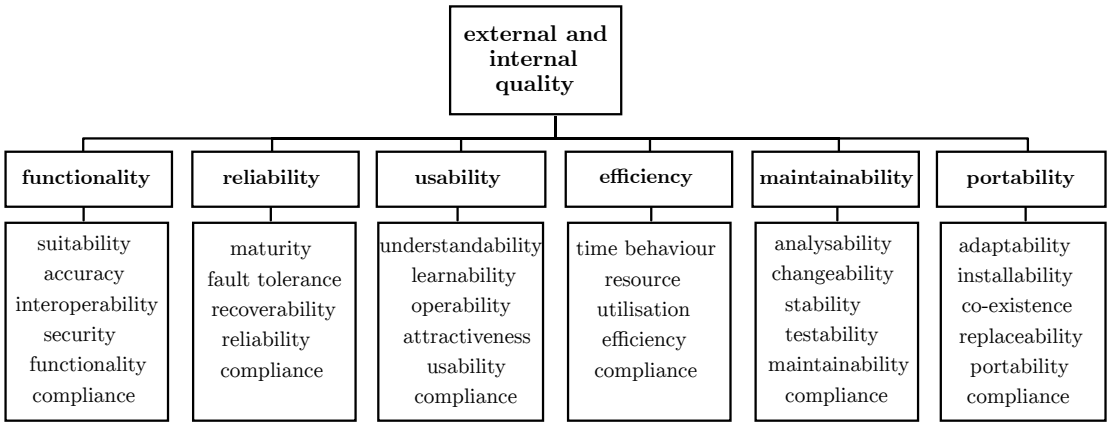


Figure 3.2.: The quality model as defined in the ISO/IEC 9126-2001 [2001]

Table 3.1.: Descriptions of the high-level quality characteristics according to the ISO/IEC 9126-2001 [2001]

Quality Characteristic	**Description**
Functionality	The capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions.
Reliability	The capability of the software product to maintain a specified level of performance when used under specified conditions.
Usability	The capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions.
Efficiency	The capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.
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.
Portability	The capability of the software product to be transferred from one environment to another.

- ***P**erformance*: Response time, resource consumption.
- ***S**upportability*: Can it be extended, adapted, corrected?

The extension of FURPS into **FURPS+** includes other quality concerns, which are not mentioned in the acronym like:

- *Design requirements*: Restrictions on the system design process e.g. which technologies to use, time taken to develop, or overall budget.
- *Implementation requirements*: Required standards, implementation language, policies for database integrity, resource limits, or operation environments.
- *Interface requirements*: External systems with which a system must interact, constraints on formats, timings, or other factors used by such an interaction.
- *Physical requirements*: Material, shape, size, or weight of the hardware.

Dromey's Quality Model Dromey [1995] proposes a product-based quality model in which each software product has specific properties and every property has corresponding quality attributes. The classification focuses on the relation between quality attributes and sub-attributes and connects product properties with quality attributes.

Robertson & Robertson Robertson and Robertson propose a requirements specification template called *Volere* [Robertson and Robertson, 1995] in their book [Robertson and Robertson, 2012]. It contains a classification scheme and checklist for requirements—for functional, non-functional and also other requirements, e.g., *project drivers*, *project constraints* and *project issues*.

The authors structure requirements into the five requirement types: *project drivers*, *project constraints*, *functional requirements*, *project issues*, and *non-functional requirements*. For the classification of NFRs, 8 major groups of NFR types are used. These groups are not fixed and can be changed based on the needs of the requirements engineers or the software project.

The authors pay special attention on the measurability of requirements and propose to use a fit criterion, i.e., how can the requirement be fulfilled, even if the requirement is fuzzy and imprecise?

Faceted Classification (Glinz) Glinz proposes in 2005 a classification of requirements according to four facets: kind, representation, satisfaction and role [Glinz, 2005].

Kind The kind describes the concerned matter of a requirement, e.g. *function*, *data*, *performance*, *specific quality* or *constraint*.

Representation The representation describes how a requirement is represented and goes hand in hand with the way how it can be verified. He lists the following forms of representation together with types of verification (list adapted from Glinz [2005]):

Form	Definition	Type of verification
Operational	Specification of operations or data	Review, test, or formal verification.
Quantitative	Specification of measurable properties	Measurement (at least on an ordinal scale).
Qualitative	Specification of goals	No direct verification. Either by subjective stakeholder judgement, by prototypes, or indirectly by goal refinement or derived metrics.
Declarative	Description of a required feature	Review

Satisfaction The satisfaction of requirements is either *hard*, that is, it is fulfilled or not, or *soft*, i.e., the degree of satisfaction is measured on a scale which is at least ordinal.

Role The role reflects the intended use of the requirement. If it is used for properties of the system, the requirement is *prescriptive*. If it is used for facts or rules in the environment, it is *normative*. For example, specific tax formulas are normative requirements. These are also sometimes called business requirements. If it is used to describe interactions between the actors and the systems, it is *assumptive*. It is assumptive, as the real actions and interactions taken by users can only be assumed and not directly predicted.

Glinz argues that, in contrast to the traditional fuzzy notion of non-functional requirements, his classification clearly separates the concerns which allows to characterize a requirement more precisely.

Concern-based Classification (Glinz) In 2007, Glinz [Glinz, 2007] discusses three problems with the notion of NFRs and revises his facets-based classification. He argues, that, although his previous classification solves these problems, there is a need in practice for a differentiation between functional and non-functional requirements. Moreover, he states that NFRs must be able to be sub-classified "in a clear and comprehensible way".

He presents a new classification that is based on so-called *concerns*. A concern is defined as *"a matter of interest in a system. A concern is a functional or behavioral concern if its matter of interest is primarily the expected behavior of a system or system component in terms of its reaction to given input stimuli and the functions and data required for processing the stimuli and producing the reaction. A concern is a performance concern if its matter of interest is timing, speed, volume, or throughput. A concern is a quality concern if its matter of interest is a quality of the kind enumerated in ISO/IEC 9126"* [Glinz, 2007].

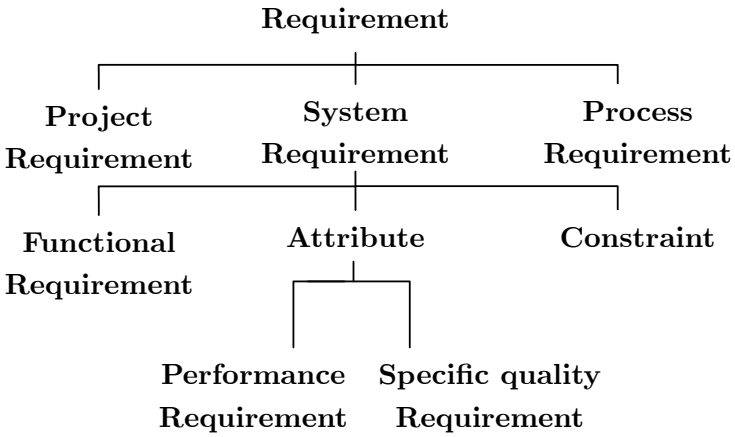


Figure 3.3.: Concern-based: Hierarchy (adapted from Glinz [2007])

Table 3.2.: Classification Rules (adapted from Glinz [2007])

#	**Question** Was this requirement stated because we need to specify...	**Result**
1	... some of the system's behavior, data, input, or reaction to input stimuli - regardless of the way how this is done?	Functional
2	... restrictions about timing, processing or reaction speed, data volume, or throughput?	Performance
3	... a specific quality that the system or a component shall have?	Specific quality
4	... any other restriction about what the system shall do, how it shall do it, or any prescribed solution or solution element?	Constraint

As shown in Figure 3.3, Glinz classifies requirements into *project*, *system*, and *process* requirements. System requirements are further classified into *functional requirements*, *attribute*, and *constraint*. Finally, attributes are classified into *performance requirements* and *specific quality requirements*.

Glinz further defines a functional requirement as a requirement that pertains to a functional concern, a performance requirement as a requirement that pertains to a performance concern, a specific quality requirement as a requirement that pertains to a quality concern other than the quality of meeting the functional requirements, a constraint as a requirement that constraints the solution space beyond what is necessary for meeting the given functional, performance, and specific quality requirements, and finally, an attribute as a performance requirement or a specific quality requirement.

He further provides classification rules that allow to classify a given requirement in the taxonomy shown in Figure 3.3. Table 3.2 shows these rules.

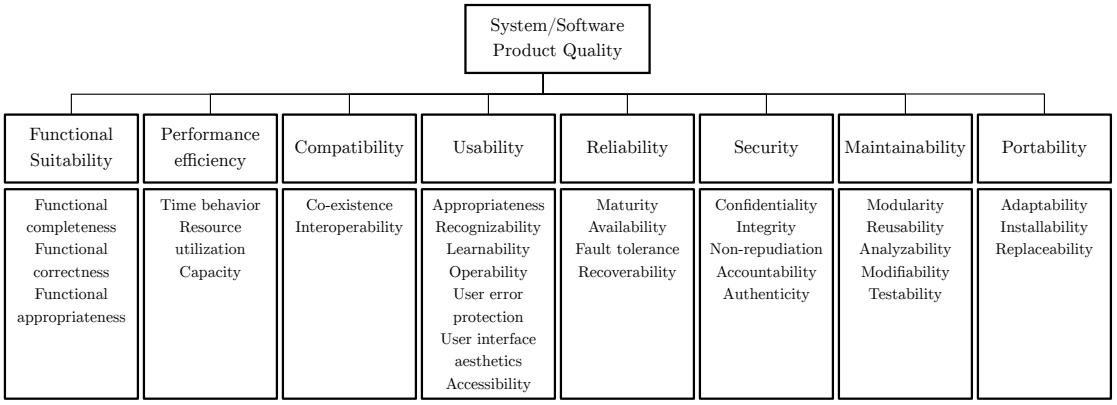


Figure 3.4.: Product quality model (adapted from the ISO/IEC 25010-2011 [2011])

ISO/IEC 25010-2011 [2011] The ISO/IEC 25010-2011 [2011] standard is a major revision of ISO/IEC 9126-2001 [2001]. Instead of three different quality models, the revised standard combines them to two: system/software product quality (replacing internal and external quality) and quality in use.

Each of the quality models has characteristics, sub-characteristics and a set of quality properties. Figure 3.4 shows the product quality model. The quality properties have to be associated with quality measures to assure that a specific sub-characteristic is covered and measurable. The standard states that the hierarchy of sub-characteristic only reflects typical quality concerns but is not exhaustive.

IREB Glossary The International Requirements Engineering Board (IREB e.V.) was founded in 2007 and is composed of independent requirements engineering experts. These experts have created a curriculum for the domain of requirements engineering and, based on the curriculum, developed a certificate, the Certified Professional for Requirements Engineering (CPRE). The curriculum contains a glossary of requirements engineering terminology [Glinz, 2014] which provides definitions for 128 central terms of requirements engineering.

The IREB Glossary basically uses the classification introduced by Glinz in 2007.

Broy Broy [2016] provides a systematic requirements categorization according to three fundamental views:

- **Black-box Interface View:** In this view, the functionality of the system is specified by means of the interface and interface behavior in terms of the interaction over the system boundaries.
- **(Logical) Sub-system Structuring View:** This view contains logical and probabilistic properties at the level of architecture—including state views of state spaces and state transitions. This view is captured by architecture and architectural behavior or by state and state-transition behavior.

- **Representation View:** This view includes technical, physical, syntactical representation, structuring, and implementation details.

For behavior, he further distinguishes between logical behavior in terms of sets of patterns of interaction and probabilistic behavior in terms of the probabilities for the patterns of interaction. In Section 2.3, a comprehensive discussion on the categorization is provided.

3.1.3. Discussion on Requirements Categorizations

In this section, we will report on the influential work of Glinz with respect to the notion of NFRs. Glinz [2007] performs in his work "On non-functional requirement" a comprehensive review on existing definitions of QRs and analyzes problems with these. He highlights three different problems with the current definitions:

Definition Problem The definition problem pinpoints the terminological and conceptual discrepancies in the various definitions. Basically, all the definitions build on the following terms *property* or *characteristic*, *attribute*, *quality*, *constraint* and *performance*. However, the terms themselves are equally fuzzy and there is no consensus about the concepts that these terms denote. Even more severe, there are also cases where the meaning of the terms is not clear as they are used without a definition or clarifying example.

Classification Problem The classification problem pinpoints the differences in the concepts for sub-classifying QRs. According to Glinz, more classification problems arise due to mixing three concepts that should better be separated: the concept of kind (should a given requirement be regarded as a function, a quality, a constraint, etc.), of representation (see below), and of satisfaction (hard vs. soft requirements).

Representation Problem The representation problem pinpoints that the notion of an QR is representation dependent. Glinz further provides examples of requirements that can be classified differently depending on the representation. A second representational problem is the lack of consensus where to document QRs. Some of the authors recommend to document the QRs separated from functional requirements, some recommend to document them together.

3.2. Types of Quality Requirements

Mairiza et al. [2010] perform a literature review on QRs, investigating the notion of QRs in the software engineering literature to increase the understanding of this complex and multifaceted phenomenon. They analyzed the three dimensions (i) definition and terminology, (ii) types, and (iii) QRs in various types of systems and application domains. Amongst others, they found about 114 different types of QRs, listed in Table 3.3. They further analyzed whether these types of QRs (i) have a definition and attributes, (ii) just have a definition, or (iii) neither have a definition nor attributes. According to their

analysis, they found that 53.51% of the types of QRs are without a definition and without attributes, 26.32% just have a definition and only 20.18% are defined with attributes. As a result of a frequency analysis, they found that the five most frequently mentioned types of QRs in literature are *performance*, *reliability*, *usability*, *security*, and *maintainability* (in that order).

This comprehensive list shows that there are many types of QRs that are discussed in literature. However, as with the requirements categorizations discussed above, there is a definition problem of the individual types of QRs: Their analysis shows that more than half of the types are presented in literature without a proper definition or attributes. This further strengthens our confidence that we need a structured approach for defining quality attributes based on literature.

3.3. Requirement Categorizations in Practice

In this section, we report on related work on requirements categorizations and their implications in practice.

One of the first studies that analyzed how to systematically deal with QRs in software development was conducted by Chung and Nixon [1995]. They argue that QRs are often retrofitted in the development process or pursued in parallel with, but separately from, functional design and that an ad-hoc development process often makes it hard to detect defects early. They perform three experimental studies on how well a given framework [Mylopoulos et al., 1992] can be used to systematically deal with QRs. They conclude that from their perspective as framework developers and users, the framework was helpful to represent and use a large number of QR specific concepts. Using the representations, they were able to consider design alternatives and their tradeoffs with respect to conflicts and synergy among the QRs, to determine the effect of each design decision, and to justify the decision with the needed design rationale.

Svensson et al. [2009] perform an interview study on how QRs are used in practice. Based on their interviews, they found that there is no QR-specific elicitation, documentation, and analysis, that QRs are often not quantified and, thus, difficult to test, and that there is only an implicit management of QRs with little or no consequence analysis. Furthermore, they found that at the project level, QRs are not taken into consideration during product planning (and are thereby not included as hard requirements in the projects) and they conclude that the realization of QRs is a reactive rather than proactive effort.

Borg et al. [2003] analyze via interviews how QRs are handled in practice by the example of two Swedish software development organizations. They found that QRs are difficult to elicit because of a focus on functional requirements, they are often described vaguely, are often not sufficiently considered and prioritized, and they are sometimes even ignored. Furthermore, they state that most types of QRs are difficult to test properly due to their nature, and when expressed in non-measurable terms, testing is time-consuming or even impossible.

Table 3.3.: Comprehensive list of types of QRs adapted from Mairiza et al. [2010], in alphabetic order.

Accessibility, Access Control	Controllability	Learnability	Scalability
Accountability	Correctness	Legibility	Security, Control and Security
Accuracy	Customizability	Likeability	Self-Descriptiveness
Adaptability	Debuggability	Localizability	Simplicity
Additivity	Decomposability	Maintainability	Stability
Adjustability	Defensibility	Manageability	Standardizability, Standardization, Standard
Affordability	Demonstrability	Maturity	Structuredness
Agility	Dependability	Measurability	Suitability
Analyzability	Distributivity	Mobility	Supportability
Anonymity	Durability	Modifiability	Survivability
Atomicity	Effectiveness	Nomadicity	Susceptibility
Attractiveness	Efficiency, Device Efficiency	Observability	Sustainability
Auditability	Enhanceability	Operability	Tailorability
Augmentability	Evolvability	Performance, Efficiency, Time or Space Bounds	Testability
Availability	Expandability	Portability	Traceability
Certainty	Expressiveness	Predictability	Trainability
Changeability	Extendability	Privacy	Transferability
Communicativenes	Extensibility	Provability	Trustability
Compatibility	Fault, Failure Tolerance	Quality of Service	Understandability
Completeness	Feasibility	Readability	Uniformity
Complexity, Interacting Complexity	Flexibility	Reconfigurability	Usability
Composability	Formality	Recoverability	Variability
Comprehensibility	Functionality	Reliability	Verifiability
Comprehensiveness	Generality	Repeatability	Versatility
Conciseness	Immunity	Replaceability	Viability
Confidentiality	Installability	Replicability	Visibility
Configurability	Integratability	Reusability	Wrappability
Conformance	Integrity	Robustness	
Consistency	Interoperability	Safety	

Ameller et al. [2012] perform an empirical study based on interviews around the question *How do software architects deal with QRs in practice?* They found that QRs were not often documented, and even when documented, the documentation was not always precise and usually became desynchronized. Furthermore, they state that QRs were claimed to be mostly satisfied at the end of the project although just a few classes were validated. With respect to model-driven development, Ameller et al. [2010] show that most model-driven development (MDD) approaches focus only on functional requirements and do not integrate QRs into the MDD process. They further identify challenges to overcome in order to integrate QRs in the MDD process effectively. Their challenges include *modeling of QRs at the PIM-level*, which includes the question *which types of QRs are most relevant to the MDD process?* According to Ameller et al. [2010], the few MDD approaches that support the modeling of QRs can be classified into approaches that use UML extensions [Fatwanto and Boughton, 2008; Wada et al., 2010; Zhu and Liu, 2009] or a specific metamodel [Gönczy et al., 2009; Kugele et al., 2008; Molina and Toval, 2009] to model QRs.

In all of the approaches, functional requirements and QRs are modeled separately. Damm et al. [2005] suggest to overcome this separation and propose a so-called rich component model based on UML that integrates functional and quality requirements in a common model. Similar approaches exist for specific classes of QRs (e.g., for availability [Junker and Neubeck, 2012]).

All these studies highlight, so far, that QRs are not integrated in the software development process and furthermore that several problems are evident with QRs.

3.4. Relation to this Dissertation

We base the motivation of this dissertation on the substantial discussion in literature on how to categorize requirements. Therefore, we initially provided a historical sequel of requirements categorizations and discussed the differences of current categorizations.

In Chapter 4, we investigate how practitioners handle requirements. We found many reasons why practitioners distinguish between quality and functional requirements that are based predominantly on some conventional wisdom on QRs, which might be based on the long-lasting discussion about QRs in literature. Furthermore, in Chapter 5, we analyze the adequacy of Broy's requirements categorization [Broy, 2016]. We use the ISO/IEC 9126-2001 [2001] to classify requirements found in practice and discuss the adequacy of Broy's classification. Finally, in Chapter 7 we provide an approach for defining, specifying, and integrating quality requirements based on a system model and apply it to the quality attributes performance and availability.

In general, there is a confusion and disagreement about QRs in literature. Thus, we discussed empirical studies that try to draw a picture of the current state of the practice in the last section. All these studies highlight, that QRs are not integrated in the software development process and furthermore that several problems are evident with QRs. We use this as a basis for the motivation for the dissertation and summarize and critically discuss the implications of requirements categorizations in practice and sketch how to

overcome deficiencies associated with QR in practice in Chapter 6. Furthermore, in Chapter 4 we performed an empirical study on how practitioners handle requirements and in Chapter 5, we perform a document analysis of requirement specifications in practice. We were able to identify similar problems and found that many reasons are grounded in conventional wisdom about QRs. This strengthens our confidence that we need to further investigate this topic.

"True, how could they see anything but the shadows if they were never allowed to move their heads?"

— Plato – The Allegory of the Cave

Chapter 4

An Investigation of How Practitioners Handle Requirements

Parts of this chapter have been previously published in the following publications:

- Eckhardt, J., Vogelsang, A., and Mendéz Fernández, D. (2016d). On the Distinction of Functional and Quality Requirements in Practice. In *Proceedings of the 17th International Conference on Product-Focused Software Process Improvement (PROFES)*, pages 31–47 (full paper, research track, 16 pages)

We still have little knowledge about to which extent the distinction between FRs and QRs makes sense from a practical perspective and what are the reasons for and the consequences of this distinction. To this end, we report in this chapter on a survey we conducted with 109 practitioners to explore whether and, if so, why they handle requirements labeled as FRs differently from those labeled as QRs. We additionally asked for consequences of this distinction w.r.t. the development process.

Our results indicate that the development process for requirements of the two classes strongly differs (e.g., in testing). We identified a number of reasons why practitioners do (or do not) distinguish between QRs and FRs in their documentation and we analyzed both problems and benefits that arise from that. We found, for instance, that many reasons are based on expectations rather than on evidence. Those expectations are, in fact, not reflected in specific negative or positive consequences per se. It therefore seems

more important that the decision whether to make an explicit distinction or not should be made consciously such that people are also aware of the risks that this distinction bears so that they may take appropriate countermeasures.

4.1. Context: Requirements Categorizations in Practice

In literature (e.g., [ISO/IEC/IEEE 29148-2011, 2011; Pohl, 2010; Robertson and Robertson, 2012; Sommerville and Kotonya, 1998; Van Lamsweerde, 2001]), requirements are often categorized in *functional requirements (FRs)*, *quality requirements (QRs)*, and *constraints*. FRs are characterized as "things the product must do" contrasting QRs as "qualities the product must have" and constraints as "organizational or technological requirements". Although this categorization is common sense to some degree, there are still debates about the precision of the categories (e.g., [Glinz, 2007]). There are other academic groups that suggest to rather distinguish between *behavior* (e.g., response times) and *representation* (e.g., programming languages) [Broy, 2016].

In a previously conducted study [Eckhardt et al., 2016c], we analyzed 11 requirements specifications from industrial environments with a particular focus on requirements labeled as "quality". We found out that (i) there is a distinction between QRs and FRs in the documentations, and that (ii) many requirements labeled as QR actually describe system behavior and, thus, could also be labeled as FR. However, our previous investigation focused on analyzing artifacts after the fact and we still have little knowledge about what difference it makes in a development process if a requirement is labeled as FR or as QR and what the resulting consequences are.

In response to this question, we report in this chapter on a survey we conducted with 109 practitioners to explore whether and, if so, why they consider requirements labeled as FRs differently from those labeled as QRs (or NFRs) as well as to disclose resulting consequences for the development process. In particular, we contribute:

1. a quantification of company practices regarding the style of documenting functional and quality requirements,

2. a list of reasons why practitioner do or do not document FRs and QRs separately,

3. a list of consequences for the two styles of documentation that helps engineers to make conscious decisions.

The remainder of the chapter is structured as follows: In Section 4.2, we state our research questions and present our questionnaire in Section 4.3. Then, in Section 4.4, we present the results before we discuss the implications of our results in Section 4.5. In Section 4.6, we discuss the limitations and threats of our study. Finally, in Section 4.7, we report on related work, before concluding our study in Section 4.8.

4.2. Research Objective

The goal of this study is to understand whether practitioners consider product-related requirements labeled as FR differently from those labeled as QR. We are further interested in the reasons for this distinction and the resulting consequences for the development process. We derive the following research questions:

RQ1 Do practitioners handle FRs and QRs differently? In this RQ, we want to analyze whether QRs are documented in practice, whether there is a distinction in the documentation, and whether this distinction makes a difference in the development process. To this end, we formulate the following sub-RQs:

- **RQ1.1 Do practitioners differentiate between QRs and FRs in the documentation?** We want to know whether the accepted categorization of product-related requirements as FRs or QRs is reflected in the style of documentation as used in practice.
- **RQ1.2 To what extent do development activities for QRs differ from activities for FRs?** A possible consequence of a requirement categorization is that different categories of requirements are handle differently in the development process. We want to investigate whether this is the case in practice and how this is influenced by the style of documentation.

RQ2 What are reasons for distinguishing or not distinguishing between QRs and FRs in the documentation? While categorizations only provide definitions, we are interested in the underlying reasons that lead practitioners to distinguish or not distinguish between QRs and FRs in the documentation.

RQ3 What are positive and negative consequences of distinguishing or not distinguishing QRs and FRs in the documentation? A decision for or against a separate documentation may have positive or negative consequences that practitioners should be aware of.

4.3. Research Methodology

In the following, we introduce the methodology applied in our study. Our goal was to reach out to a broad spectrum of practitioners and capture their perceptions of their own project environments. To this end, we used (online) survey research as our main vehicle. We intentionally designed the survey such that respondents required as little effort as possible to complete it; we kept the number of questions at a minimum, the instrument was self-contained and it included all relevant information. We further limited the response types to numerical, Likert-scale, and short free form answers as suggested by Kitchenham and Pfleeger [2008]. As a validation of our instrument and its alignment with the audience, we piloted the survey with three practitioners, who completed the survey and afterwards participated in an interview, where questions and answers where checked for misunderstandings.

In the following, we describe the particularities of our subject selection, before discussing the data collection and instrument, and the data analysis.

4.3.1. Subject Selection

We deliberately targeted practitioners who work with requirements. This includes practitioners who write requirements (e.g., *requirements engineers*) but also practitioners whose work is based on requirements (e.g., *developers* or *testers*), and also practitioners who manage projects or requirements. Our survey was further conducted anonymously. Since we were not able to exactly control who is answering the survey, it was especially important to follow the advice of Kitchenham and Pfleeger [2008] on the need to understand whether the respondents had enough knowledge to answer the questions in an appropriate manner. For this, we excluded data from respondents who answered that they do not use requirements specifications at all, or respondents who stated that they did not know how requirements are handled in their company. We finally offered respondents the chance to leave an email address if they were interested in the results of the survey.

4.3.2. Data Collection and Instrument

We started our data collection on February 4th, 2016 and closed the survey on February 22nd, 2016. For inviting practitioners to participate, we did not select a specific closed group of practitioners but, instead, contacted as many practitioners as possible via the authors' personal contacts from previous collaborations, via public mailing lists such as *RE-online*, and via social networks. In the following, we introduce the main elements of our instrument used. The full instrument can be taken from our online material[7].

Demographics

We collected a set of demographic data from the respondents to interpret and triangulate the data with respect to different contexts of the respondents. The demographic data included the role of the participant, the experience, the company's size, the typical project size, the geographical distribution of project members, the paradigm of their applied development process (on a scale from agile to plan-driven), the industrial sector, the type of developed systems, and the role of the requirements specification within the company. To better understand the participant's focus and project context, we additionally asked respondents for the importance of different types of QRs in their projects. The respondents were asked to assess the importance of quality factors[8] taken from ISO/IEC 25010-2011 [2011] for their typical projects on a 5-point Likert scale.

[7] `http://www4.in.tum.de/~eckharjo/SurveyResults.zip`

[8] These were functional suitability, performance/efficiency, compatibility, usability, reliability, security, maintainability, and portability.

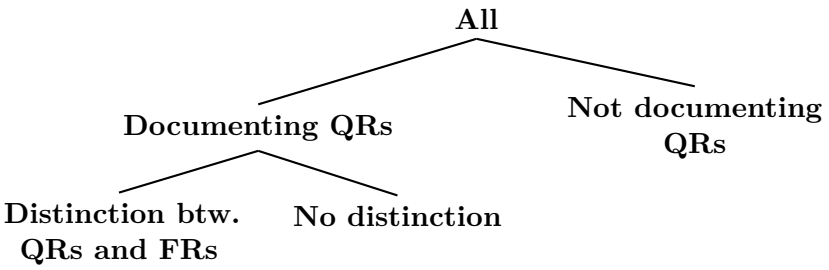


Figure 4.1.: Categorization of respondents by their style of documenting QRs.

Practices of Handling QRs

As a first step towards comparing different practices for handling QRs, we asked the respondents how strongly development activities differ between QRs and FRs in the phases *requirements engineering*, *architecture/design*, *implementation*, and *testing*. As a follow up, we provided a free form text field and asked the respondents to explain the differences in detail.

We were especially interested in the question whether it makes a difference for the development process if project participants distinguish between QRs and FRs and how this distinction is documented. Therefore, we asked the respondents two conditional questions. First, we asked whether QRs are explicitly documented in their projects. If this was the case, we asked whether the respondents explicitly distinguish between QRs and FRs in the documentation, i.e. whether they are labeled differently (e.g., some requirements are labeled as *performance* or *maintainability*) or documented in different sections (e.g., special sections for *performance* or *maintainability*). The answers to these questions categorize the responses into three groups (see also Figure 4.1).

Problems/Benefits of Current Practices:

Given the categorization into the three groups, we asked our respondents for specific reasons why they do or do not distinguish between QRs and FRs. Additionally, we asked for benefits and problems that arise from the way they consider QRs (i.e., not documenting QRs, mixing QRs and FRs in the documentation, or distinguishing between QRs and FRs in the documentation). For these questions, we provided free form text fields to be filled out by the respondents.

4.3.3. Data Analysis

Our data analysis constitutes a mix of descriptive statistics and qualitative text analysis. To answer RQ1, we analyzed in particular the answers that the respondents provided for the following survey questions: (i) *Are QRs documented in your typical projects*, (ii) *In the documentation (e.g., in a requirements specification), do you distinguish between QRs and FRs*, (iii) *Considering the following phases, how much do the activities for handling QRs differ from those for FRs*, and (iv) *Considering your work, for what activities does it*

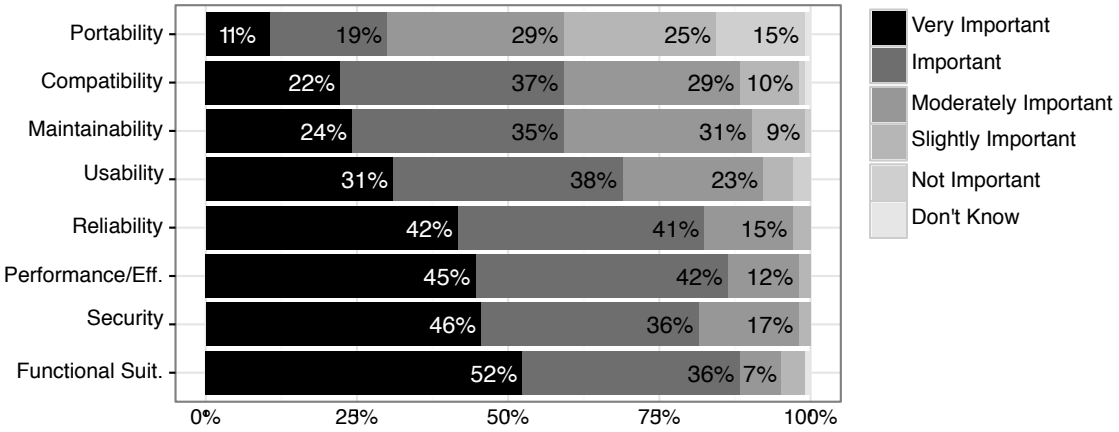


Figure 4.2.: How important do you consider the following types of QRs for your typical projects?

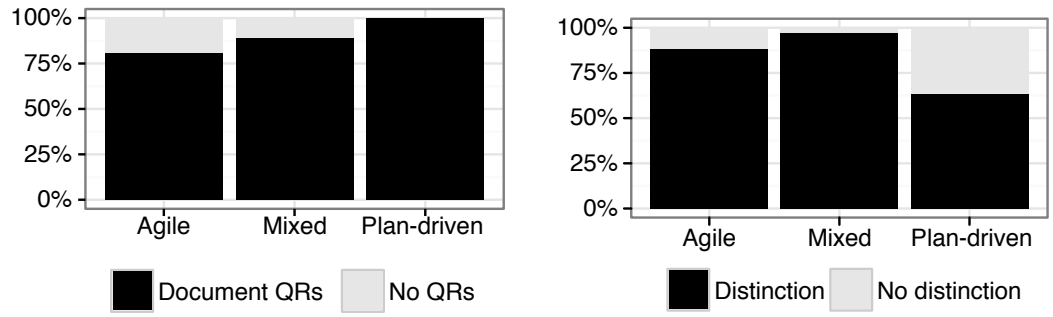
make a difference if you consider an QRs vs. an FR. For RQ1.1 and RQ1.2 we analyzed the results of the first, second, and third question, respectively. As the answers for the fourth question are open, we analyzed the answers in detail to provide more insights in the activities and the differences.

To answer RQ2 and RQ3, we analyzed the data our respondents provided for the following survey questions: (i) *Are there specific reasons why you do (or do not) distinguish between QRs and FRs in the documentation*, (ii) *Do you experience negative consequences in your current work that result from distinguishing (not distinguishing) between QRs and FRs in the documentation*, and (iii) *Do you experience positive consequences in your current work that result from distinguishing (not distinguishing) between QRs and FRs in the documentation*. The answers to the questions are free text answers. To analyze the results, we coded the provided answers in pairs of researchers to assemble a conceptual model of reasons and consequences for distinguishing between QRs and FRs in practice. The qualitative coding technique was chosen as recommended by (Straussian) Grounded Theory [Stol et al., 2016], but differs in that the central categories were previously defined following our research questions. To visualize our results from the text analysis, we used cause-effect diagrams (also known as Ishikawa diagrams).

4.4. Study Results

4.4.1. Sample Characterization

In total, 283 people clicked on the link to our survey, 172 started the survey (61%), and 109 completed it (39%). From these 109 respondents, we excluded 6 as they matched our exclusion criteria. The respondents seem quite experienced as 93% stated that they have more than 3 years of experience with requirements, 5% one to three years,



(a) Do you document QRs?

(b) Do you distinguish btw. QR & FR?

Figure 4.3.: Relation between process paradigm and the style of documenting QRs

and only 2% with less than a year. Furthermore, a majority of the respondents work in large companies: 57% work in companies with more than 2000 employees, 25% in companies with 250–2000 employees, and 17% in companies with less than 250 employees. However, typical projects of the respondents showed a variety of small to large projects: 24% stated that in a usual project in their company up to 10 people are involved, 46% that 11–50 people are involved, 24% that more than 50 people are involved, and 6% did not know. Most of the respondents (59%) answered that their team is distributed over multiple locations in more than one country, 23% that the team is distributed over multiple locations but in one country, and 17% that all team members are in one location. The employed process paradigm is balanced between agile and plan-driven: 41% of the respondents answered that their development process is rather agile, 21% that it is rather plan-driven, 37% that it is mixed, and 1% did not know. The type of systems the respondents develop is quite balanced (except for consumer software): 24% develop embedded systems, 37% business information systems, 5% consumer software, and 34% hybrid systems. Most of the respondents use requirements specifications for in-house development (57%), 23% create them and an external company is responsible for the development, and 19% are subcontractors using requirements specifications (e.g., as basis for development or testing). Figure 4.2 shows how our respondents ranked the importance of different quality factors w.r.t. their daily work on a five point Likert scale.

4.4.2. RQ1: Handling of QRs in Practice

RQ1.1: Do practitioners differentiate between QRs and FRs in the documentation?

88% of the respondents answered that they document QRs in their projects, while 12% answered that they do not document QRs at all. We contextualized this distribution w.r.t. the process paradigm the respondents use in their projects. Figure 4.3a shows that all respondents with a plan-driven process document QRs, while in agile processes only 77% document QRs.

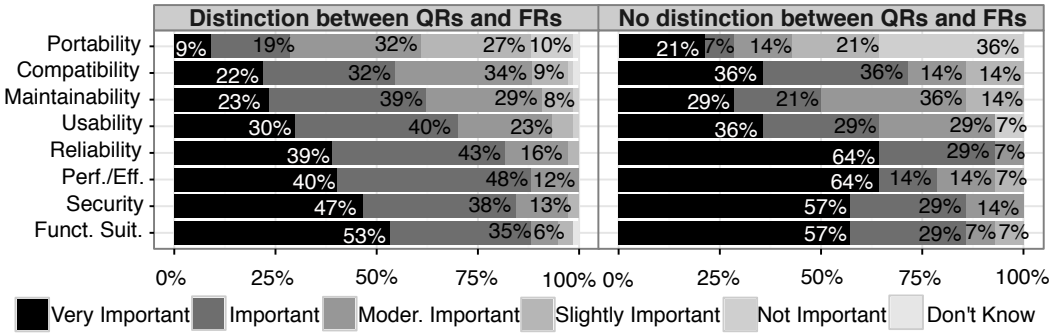


Figure 4.4.: Relation btw. importance of quality attributes and style of documentation

From the respondents who document QRs (91 in total), 85% answered that they distinguish between QRs and FRs in the documentation and 15% answered that they do not. We also contextualized this distribution w.r.t. the process paradigm. Figure 4.3b shows that a higher percentage of the respondents in agile processes distinguish between QRs and FRs compared with respondents in plan-driven processes. As a second contextualization, we analyzed the importance of quality factors w.r.t. the style of documentation. Figure 4.4 shows how the respondents ranked the importance of different quality factors for their daily work on a five point Likert scale. *Reliability* and *Performance/Efficiency*, for example, stand out as they are considered more important by participants who do not distinguish between QRs and FRs.

RQ1.2: To what extent do development activities for QRs differ from activities for FRs?

Figure 4.5 shows how the respondents ranked the difference in the phases requirements engineering, architecture/design, implementation, and testing on a three point Likert scale. As a contextualization, we analyzed whether there is a difference in how respondents rank the difference in the development phases w.r.t. whether they do or do not distinguish between QRs and FRs (Figure 4.6). The figure shows that the phase architecture/design was reported to differ stronger by respondents who distinguish between QRs and FRs.

To further detail this response, Table 4.1 shows exemplary statements that respondents gave explaining the differences in the development activities. According to the answers, there is a different maturity of the processes for treating FRs vs. QRs (see Statement A). Furthermore, when it comes to project planning, FRs are planned in detail but QRs are considered in an unplanned way and only documented on a high-level (see Statement B). In testing, there are approaches for deriving test cases from FRs but none for deriving them from QRs (see Statement C). Moreover, different stakeholders are involved in testing QRs vs. FRs (see Statement D). In architecture and design, QRs need to be considered early in the project as they have a high impact on the architecture. In contrast to this, it is sufficient to consider FRs at an abstract level in early stages (see Statement E). In the implementation, QRs need to be monitored continuously, whereas FRs can be

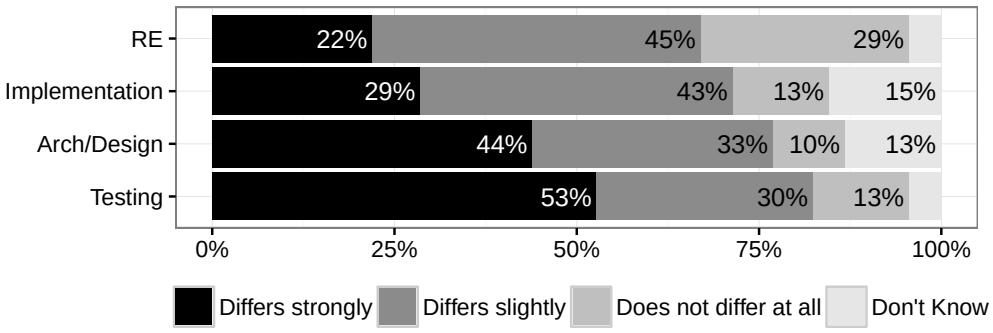


Figure 4.5.: Considering the following phases, how much do development activities for QRs differ from activities for FRs?

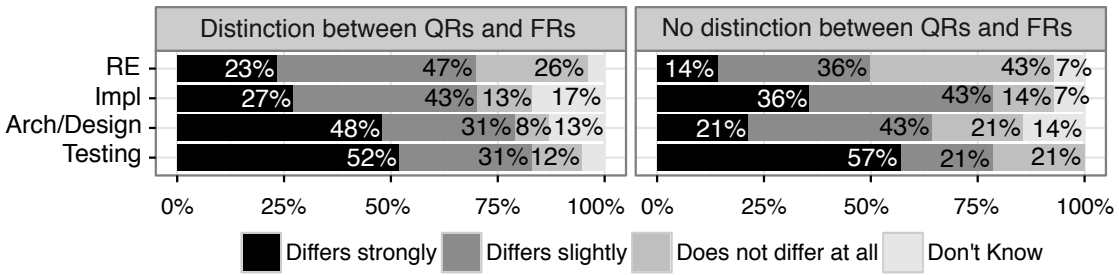


Figure 4.6.: Relation between process differences and style of documentation

Table 4.1.: Exemplary answers about differences in the development process

#	Phase	Answer
A.	General	*"[QRs] are usually treated less transparent: not clearly documented, not explicitly tested, but somehow considered in RE, design and coding as common sense background, e.g., in terms of [QRs] considering IT security, performance or reliability."*
B.	General	*"FRs are documented and planned in high detail [...] Working on [QRs] are often unplanned activities and only high level documented."*
C.	Test	*"Test cases for FR[s] can quite easily [be] derived from functional models or textual requirements [... but there is no] method for deriving test cases from [QRs]."*
D.	Test	*"Test planning, preparation and execution for [QRs] are handled by different stakeholders ([QRs] are [. . .] strongly architecture related) and personnel (performance and load tests are performed by specialists usually not part of the project team)."*
E.	Arch.	*"[QRs] are often architectural drivers and therefore have to be evaluated and considered very early in the project when defining the architecture. Whereas in an early stage of the project a more abstract view on the functional requirements is sufficient."*
F.	Impl.	*"[QRs] require continuous monitoring, as achievements (e.g., performance) may degrade during implementation."*
G.	RE	*"[In contrast to FRs,] [QRs] can be negotiated, if they are technically not reachable."*

implemented successively (see Statement F). In requirements engineering, FRs are more fixed than QRs as QRs can be negotiated with the customer while FRs usually cannot (see Statement G).

RQ1: Summary of Results
Most (88%) of the respondents document QRs in their projects and from these, 85% distinguish between QRs and FRs in the documentation. Furthermore, testing is the activity where handling QRs vs. FRs differs most.

4.4.3. RQ2: Reasons for Distinguishing QRs and FRs

Figure 4.7 and 4.8 show the cause-effect diagrams for the reasons for and consequences of (not) distinguishing between QRs and FRs in practice. On the left-hand side of the diagrams, the mentioned reasons for distinguishing (Figure 4.7) or not distinguishing (Figure 4.8) between QRs and FRs are indicated. On the right-hand side of the diagrams, the mentioned consequences of the decision are shown. The upper part contains the positive consequences while the lower part contains the negative consequences. The different entries of the diagrams (e.g., *QRs have different nature* in Figure 4.7) correspond to codes that we identified in the data and their number of occurrences. Furthermore, we structured the codes in categories that are represented by the arcs in the diagram.

Reasons for Distinguishing QRs and FRs

The left-hand side of Figure 4.7 shows the resulting reasons for distinguishing between QRs and FRs. In total, 49 out of the 77 respondents (64%) that distinguish between QRs and FRs provided an answer to this open question. We identified 24 codes in the answers for this question. For clarity, we only show codes that occurred at least twice in Figure 4.7. Reasons that we coded as *QRs have different nature*, *Company Practice*, and *QRs are cross-functional* occur frequently in the category *General & Project Organization*. Furthermore, in the category *Design & Implementation* the reason *Influence the architecture* and in the category *Validation & Verification* the reason *QRs require different verification methods* also occur often.

Reasons for Not Distinguishing QRs and FRs

The left-hand side of Figure 4.8 shows the mentioned reasons for not distinguishing between QRs and FRs. In total, 7 out of the 14 respondents (50%) who do not distinguish between QRs and FRs provided an answer to this open question. We identified 8 codes in the answers for this question. Figure 4.7 shows all identified codes, which all occurred only once in the data (except for *There is no difference*).

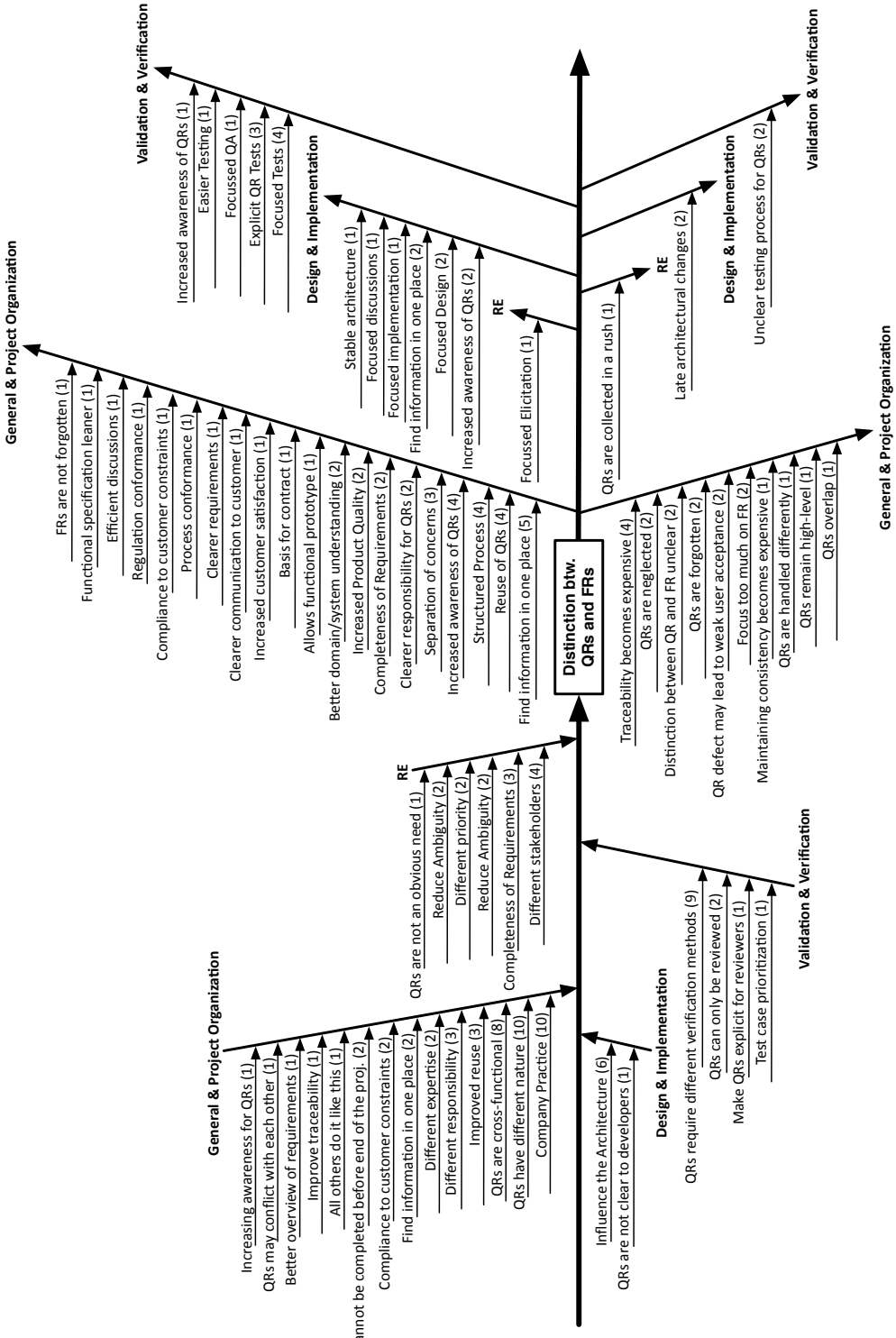


Figure 4.7.: Reasons for and consequences of distinguishing between QRs and FRs. The left-hand side shows the mentioned reasons and the right-hand side the mentioned consequences. The upper part of the right-hand side contains the positive consequences while the lower part contains the negative consequences.

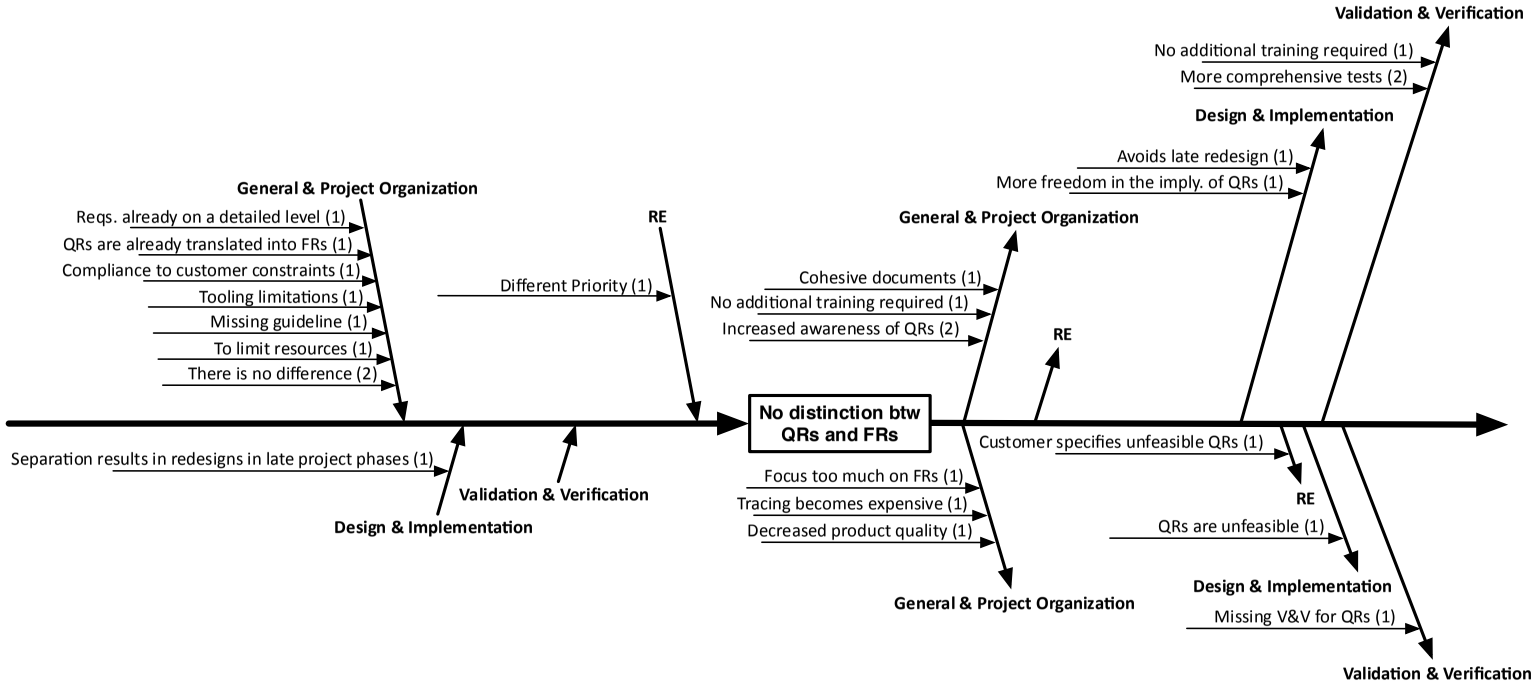


Figure 4.8.: Reasons for and consequences of not distinguishing between QRs and FRs. The left-hand side shows the mentioned reasons and the right-hand side the mentioned consequences. The upper part of the right-hand side contains the positive consequences while the lower part contains the negative consequences.

RQ2: Summary of Results
Distinction: The most frequently coded reason for distinguishing between QRs and FRs is that QRs have a different nature compared with FRs (10) and company practice (10).
No distinction: The most frequently coded reason for not distinguishing between QRs and FRs is that there is no difference between QRs and FRs (2).

4.4.4. RQ3: Benefits and Problems

Benefits and Problems of Distinguishing QRs and FRs

The right-hand side of Figure 4.7 shows the consequences of distinguishing between QRs and FRs. The upper part shows the positive consequences while the lower part shows negative consequences. In total, 45 out of the 77 respondents (58%) that distinguish between QRs and FRs provided answers to the open question about positive consequences. Regarding negative consequences, 16 out of the 77 respondents (21%) provided answers. We identified 35 codes in the answers for positive consequences and 13 in the answers for negative consequences. As shown in the diagram, the code that we identified most in the mentioned benefits is *Find information in one place* in the category *General & Project Organization*. In this category, there are also other benefits that occurred frequently (e.g., structuredness of the process, completeness of the requirements, separation of concerns, and increasing the awareness of QRs). We coded the benefit *Increased awareness of QRs* also three times in the category implementation. For validation and verification, the most frequent benefits are *Focused Tests* and *Explicit QRs Tests*. The code that we identified most in the mentioned problems is *Traceability becomes expensive*. Further problems that were mentioned are that QRs are neglected or forgotten, that the distinction between QRs and FRs is unclear and that the distinction results in a weak user acceptance. Moreover, in the category *Validation & Verification*, the problem *Missing testability* was mentioned.

Benefits and Problems of Not Distinguishing QRs and FRs

The right-hand side of Figure 4.8 shows the consequences of not distinguishing between QRs and FRs. The upper part shows the positive consequences while the lower part shows negative consequences. In total, 9 out of the 14 respondents (64%) that distinguish between QRs and FRs provided answers to the open question about positive consequences. Regarding negative consequences, 5 out of the 14 respondents (36%) provided answers. We identified 7 codes in the answers for positive consequences and 6 in the answers for negative consequences.

RQ3: Summary of Results
Distinction: The most frequently coded benefit for distinguishing between QRs and FRs is to find information on one place (5). The most frequently coded problem is that traceability becomes expensive (4).
No distinction: The most frequently coded benefit for not distinguishing between QRs and FRs is an increased awareness of QRs (2).

4.5. Discussion

Based on the results, we identified a set of insights that we discuss in the following paragraphs.

4.5.1. Different Handling of QRs and FRs

Based on our previous study [Eckhardt et al., 2016c], we expected that the majority of QRs are documented in practice and that practitioners make a distinction between QRs and FRs. The results of RQ1.1 support these expectations. Interestingly, as shown in Figure 4.3a, the only respondents who do not document QRs follow agile processes, while in a more plan-driven process, all respondents state to document QRs. This could be explained by the trend to a light-weight documentation in agile processes. In contrast to this, as shown in Figure 4.3b, in a plan-driven process, less people tend to distinguish between QRs and FRs than in a mixed or an agile process. We currently do not have an explanation for this.

Moreover, as shown in Figure 4.4, our results indicate that the importance assigned to the QR classes is dependent on whether QRs and FRs are distinguished. Especially the two QR classes reliability and performance stand out. One possible explanation for this is that reliability and performance requirements mainly describe external interface behavior of a system and, thus, are very similar to FRs [Eckhardt et al., 2016c]. Therefore, we speculate that for these classes a distinction is not necessary.

Furthermore, the results of RQ1.2 indicate that, for testing, the activities for handling QRs are very different from the activities for handling FRs. This is tune with the argument that QRs require different verification methods (see also the results of RQ2 and RQ3). Interestingly, it makes a substantial difference if we compare the differences in the phase *Architecture/Design* of respondents who make a distinction with respondents who do not make a distinction (see Figure 4.6). Since we currently do not have an explanation for this, we believe that this finding needs to be investigated further in the future.

4.5.2. Reasons for and Consequences of a Distinction

From the results presented in the previous section, we conclude that practitioners are split into two groups; one advocating a distinction between QRs and FRs and one advising against it. Interestingly, the respondents stated contrary reasons as arguments for or

against a distinction (e.g., *"Both are requirements"* vs. *"We distinguish them because they are different"*). Similarly, we found the same benefits stated by respondents of both parties: *"If you distinguish, then QRs are considered better"* vs. *"As soon as QRs are treated equally to FRs it is a clear win-win situation such that QRs get the same attention."* Additionally, our results indicate that it is not clear to practitioners what the difference between both classes of requirements actually is, even though they stated reasons, benefits, and problems of a distinction: *"Most people have problems to distinguish between them, so they mix"* or *"[Not distinguishing] avoids unnecessary confusion at the requirements authors' side. Adding the distinction QR/FR would require additional training, QS, etc. without adding value to the projects"*. Some respondents see this as a reasons why they do not distinguish between them: *"[...]There is just no real guideline how to do it"*.

The most prevalent reasons for distinguishing between QRs and FRs are in line with those that are often found in literature (e.g., QRs have a different nature and are cross-functional, influence on architecture, require different verification methods). However, we cannot underpin any of those reasons with negative consequences in the cases where QRs and FRs were not distinguished. Therefore, we conclude that there seems to be confusion about this topic in practice and handling QRs seems to be driven by expectations rather than by evidence.

In the following, we will detail and discuss some conflicting or even contradictory statements. We believe that these are topics that need to be investigated further in the future, or, in case of a clear scientific position about a topic, we need to invest more into the dissemination of the results into practice.

QR Testing – A Double-edged Sword

One of the top reasons mentioned for distinguishing QRs and FRs was the need for different verification methods (especially w.r.t. testing). Fig. 4.6 also shows that testing is the activity that differs most for QRs and FRs. When considering consequences of distinguishing between QRs and FRs in testing, we found both positive and negative. While some respondents said that a distinction leads to more focused and specialized tests for specific QRs, some also stated that a distinction leads to the fact that some QRs are not tested at all. For example, *"Performance tests are recognized as [a] key success factor by project managers"* vs. *"Main issue is how to handle the [QR] tests before product release"*. On the other hand, respondents who do not distinguish between QRs and FRs also reported positive and negative consequences regarding testing: *"[...]the mapping [of FRs to QRs] should ensure that this testing also covers [QRs]"* vs. *"[When not distinguishing,] corresponding V&V suffers"*. We conclude from this that distinguishing QRs and FRs supports the awareness for specialized tests of important QRs but, simultaneously, bears the risk of neglecting tests for less important QRs.

Company Practice – Never Change a Running Game

Another commonly stated reason for distinguishing between QRs and FRs is that this is common practice in the company or that this is required by customers. However, these reasons were almost never questioned or justified. For example, *"[. . .] Our specification template prescribes a structuring w.r.t. [QRs] and FRs"* or *"[we distinguish] as requested by the customer"*. Additionally, the respondents did not mention any positive or negative consequences that result from complying with customer constraints. We consider this as a sign of inadvertent handling of this topic. It would be interesting to ask customers to explicitly state reasons why they request a distinction of QRs and FRs.

QRs – Drivers for the Architecture

Several respondents stated that the architecture of a system is specifically influenced by QRs. For example, *"[QRs] are often architectural drivers and therefore have to be evaluated and considered very early in the project when defining the architecture"*. This was often used as an argument to distinguish between QRs and FRs: *"The separation allows architects to get a quick (and in-depth) understanding of the QRs without needing to know all the functional requirements"*. FRs, on the contrary, were considered to be more local and do not need to be fixed at the beginning of the project: *"[It is] easier to find[. . .]special FRs for developing a single use case"* or *"[. . .]in an early stage of the project a more abstract view on the functional requirements is sufficient"*. Surprisingly, some respondents stated that it has a positive impact for the implementation when QRs and FRs are not strictly distinguished: *"[QRs] and FRs are handled as features. They are not separated, which avoids the redesigns e.g., due to performance problems"* and *"[When not distinguishing,] we have much more freedom during the implementation iterations[. . .]to find solutions that fit the customers' expectations and the possibilities that come with the architecture and technology we use"*.

Awareness Matters

It seems that an increased awareness for QRs was considered as one of the most prominent benefits. Both parties claimed this as a benefit of distinguishing respectively not distinguishing between QRs and FRs: *"[Distinction] ensures that [QRs] are also in the focus"* vs. *"[Not distinguishing] helps keeping the team aware that the device does not only need to have certain features, but that these features also need to work e.g., at a high temperature"*. It seems that awareness can be increased with both strategies. The crucial point seems to be that there is a clear and explicit relation between FRs and QRs, which leads to the following observation.

Tracing – The Good, the Bad, and the Ugly

One trade-off that we found in the data is an inherent challenge that does not seem to be resolved in practice. Some respondents stated that a distinction between QRs and FRs is beneficial because it keeps associated information in one place and, thus,

supports different viewpoints on the requirements: *"People who are particularly concerned with QRs, such as architects and performance testers, find relevant information in one place"* and *"As most [QRs] apply across components, they are more easily retrieved in a separate specification"*. However, this benefit also comes with clear disadvantages considering tracing and the risk of forgetting requirements: *"Consistent documentation of relationships between FRs and [QRs] is difficult"* and *"The development team needs to be fully aware about all sources for requirements. Ostrich strategy causes a high yield of trouble"*. Respondents who do not distinguish reported on benefits regarding the cohesiveness of their specifications: *"Some documents benefit from this, as they turn more cohesive"* or *"[. . .] the feature is really ready if installed and not only 80%"*.

4.6. Limitations and Threats to Validity

We now discuss the threats to validity and mitigation measures we applied.

Participant Selection One limitation in the study is the missing lack of control over the respondents given that we distributed the survey invitation over various networks. Apart from an unknown response rate, this means that we cannot control how representative the responses are. We removed those respondents from the population that stated that they do not deal with requirements. Also, although the introductory texts explicitly stated that the survey is aimed at addressing practitioners' perspective, we cannot guarantee that all the views taken really result from practitioners.

Survey Research Further threats to the validity result from the nature of survey research. We cannot control on which basis the respondents provide their answers, the respondents might be biased, and there exists the possibility that they have misinterpreted some of the questions. We reduced the first threat by asking questions to characterize the context of the respondents. We cannot mitigate the second threat, but reduced it by conducting the survey anonymously. We minimized the third threat by conducting a pilot phase in which we tested the instrument used and the data analysis techniques applied.

Subjectivity of Coding A further major threat to validity, however, arises from the data analysis, i.e., the coding process, because coding is a creative task. Subjective views of the coders, such as experiences and expectations, might have influenced the way we coded the free text statements. A threat arises from the fact that we cannot validate our results with the respondents given the anonymous nature of our survey. We minimized this threat by coding in pairs (researcher triangulation).

Representativeness of the Codes Finally, one limitation stems from the result set itself and its expressiveness. Our focus was to collect and code practitioners' experiences on how they consider QRs. We quantified the results to get an overview of whether certain codes dominate others. However, a potentially high frequency

of codes does still not allow for conclusions on the criticality of those codes. In particular, the fact that we got more answers about reason for and consequences of a distinction between QRs and FR than for no distinction might have distorted our interpretation of the results.

4.7. Related Work

The literature on requirements categorizations is very extensive. Major contributions address categorizing non-functional requirements (e.g., [Chung and do Prado Leite, 2009; Glinz, 2007; Pohl, 2010]), of which most rely on quality (definition) models. Pohl [2010], for instance, discusses the misleading use of the term "non-functional" and argues to use "quality requirements" for product-related NFRs that are not constraints. Glinz [2007] performs a comprehensive review on the existing definitions of NFRs, analyzes problems with these definitions, and proposes a definition on his own. Mairiza et al. [2010] perform a literature review on QRs, investigating the notion of QRs in the software engineering literature to increase the understanding of this complex and multifaceted phenomenon. They found 114 different QR classes. Contributions such as those have fostered valuable discussions on the fuzzy terminology used and the concepts applied, but they did not focus on the implications of these categorizations on development processes in practice.

A broader investigation on the status quo in RE, problems in RE and root causes is taken by the *Naming the Pain in Requirements Engineering* (NaPiRE) initiative. This initiative relies, same as we do, on survey research [Méndez Fernández and Wagner, 2014] and investigates, for instance, how practitioners elicit requirements, which problems they encounter (such as unclear or not measurable non-functional requirements), and what implications this has (such as an increased effort in testing). Although survey research on RE, such as this one, investigates RE from a practical perspective, it does not take a specific view on how practitioners consider NFRs.

Chung and Nixon [1995] investigate how practitioners handle QRs. They argue that QRs are often retrofitted in the development process or pursued in parallel with, but separately from, functional design and that an ad hoc development process often makes it hard to detect defects early. They perform three experimental studies on how well a given framework [Mylopoulos et al., 1992] can be used to systematically deal with QRs. Svensson et al. [2009] perform an interview study on how QRs are used in practice. Based on their interviews, they found that there is no QR-specific elicitation, documentation, and analysis, that QRs are often not quantified and, thus, difficult to test, and that there is only an implicit management of QRs with little or no consequence analysis. Furthermore, they found that at the project level, QRs are not taken into consideration during product planning (and are thereby not included as hard requirements in the projects) and they conclude that the realization of QRs is a reactive rather than proactive effort.

Borg et al. [2003] analyze via interviews how QRs are handled in two Swedish software development organizations. They found that QRs are difficult to elicit because of a focus on FRs, they are often described vaguely, are often not sufficiently considered and

prioritized, and they are sometimes even ignored. Furthermore, they state that most types of QRs are difficult to test properly due to their nature, and when expressed in non-measurable terms, testing is time-consuming or even impossible. Ameller et al. [2012] perform an empirical study based on interviews around the question *How do software architects deal with QRs in practice?* They found that QRs were often not documented, and even when documented, the documentation was not always precise and usually became desynchronized.

In all of the investigations, FRs and QRs are treated separately, and the investigations take an observational perspective on how practitioners deal with QRs in that context. The goal of our study is to analyze whether practitioners handle FRs and QRs differently, which reasons motivate the way they consider QRs, and what consequences—positive and negative ones—this has on the development process.

4.8. Conclusions

The goal of this chapter was to reach our first objective. In particular, our goal was to understand if practitioners distinguish "quality" and "functional" requirements, and if so, why they consider requirements labeled as "functional" differently from those labeled as "quality" as well as to disclose resulting consequences for the development process. To this end, we reported in this chapter on a survey we conducted with 109 practitioners.

Our results indicate that practitioners document QRs and most of them do make an explicit distinction between QRs and FRs in the documentation. Furthermore, our data suggests that the development process strongly differs depending on a distinction between QRs and FRs, especially in interconnected activities such as testing. The rationale of practitioners is that QRs are different to FRs, i.e. they are of different nature, are cross-functional, strongly influence the architecture, and require different verification methods. Furthermore, making a distinction or not does not have negative or positive consequences per se. It therefore seems more important that the decision whether to make an explicit distinction or not should be made consciously such that people are also aware of the risks that this distinction bears so that they may take appropriate countermeasures. A distinction might, for example, be justified by specialized testing teams for specific quality attributes or by requirements that are reused between a number of projects. A direct consequence of this conscious decision is that people are also aware of the potential risks that this distinction bears (e.g., the importance of trace links between FRs and QRs to assure that QRs are not neglected).

From the results of this study, we conclude that there are several issues with requirements categorizations in practice. This is in line with existing evidence; Up until now there does not exist a commonly accepted approach for the QR-specific elicitation, documentation, and analysis [Borg et al., 2003; Svensson et al., 2009]; QRs are usually described vaguely [Ameller et al., 2012; Borg et al., 2003], remain often not quantified [Svensson et al., 2009], and as a result remain difficult to analyze and test [Ameller et al., 2012; Borg et al., 2003; Svensson et al., 2009]. Furthermore, QRs are often retrofitted in the development process or pursued in parallel with, but separately from, functional

requirements [Chung and Nixon, 1995] and, thus, are implicitly managed with little or no consequence analysis [Svensson et al., 2009]. This limited focus on QRs can result in the long run in high maintenance costs [Svensson et al., 2009].

In summary, we conclude that QRs are not (sufficiently) integrated in the software development process and furthermore that several problems are evident with QRs. In the next chapter, we empirically investigate whether a requirements categorization that is based on a system model is adequate for industrial requirements.

"Nucular... it is pronounced NUCULAR."

— Homer Jay Simpson

Chapter 5

An Analysis of the Adequacy of a Categorization based on a System Model with Respect to Requirements found in Practice

---

Parts of this chapter have been previously published in the following publications:

- Eckhardt, J., Méndez Fernández, D., and Vogelsang, A. (2015). How to specify Non-functional Requirements to support seamless modeling? A Study Design and Preliminary Results. In *Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 164–167 (short paper, research track, 4 pages)
- Eckhardt, J., Vogelsang, A., and Méndez Fernández, D. (2016c). Are Non-functional Requirements Really Non-functional? An Investigation of Non-functional Requirements in Practice. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 832–842 (full paper, research track, 10 pages)

The goal of this chapter is to analyze whether a requirements categorization based on a system model is adequate[1] for requirements found in practice. In particular, we analyze 11 requirements specifications from 5 different companies for different

application domains and of different sizes with in total 530 requirements that are labeled as "non-functional", "quality", or any specific quality attribute. Our results show that 75% of the requirements labeled as "quality" in the considered industrial specifications describe system behavior and 25% describe the representation of the system. As behavior has many facets, we further categorize behavioral QRs according to the *system view* they address (interface, architecture, or state), and the *behavior theory* used to express them (syntactic, logical, probabilistic, or timed) [Broy, 2015, 2016]. Based on this fine-grained categorization that is based on a system model, we discuss the implications we see on handling QRs in the software development phases, e.g., testing or design.

Based on the results of our study, we conclude that most requirements labeled as "quality" requirements are misleadingly declared as such, as they describe behavior of the system. This in turn means that many so-called QRs can be handled similarly to functional requirements. This contribution supports (the first part of) our hypothesis, i.e., a categorization based on a system model is applicable and adequate[1] for requirements found in practice.

The remainder of this chapter is structured as follows: In Section 5.1, we discuss issues with requirements categorizations in practice and derive the problems that we are tackling in this chapter. Then, in Section 5.2, we discuss background and related work, and, subsequently, we present our study design in Section 5.3. We report on the results in Section 5.4 and discuss the threats to validity and our mitigation strategies in Section 5.5. In Section 5.6, we provide a discussion of the overall results and their impact on theory and practice, before concluding this chapter in Section 5.7.

5.1. Context: Requirements Categorizations and their Implications in Practice

One conventional distinction between QRs and FRs is made by differentiating *how* the system shall do something in contrast to *what* the system shall do [Robertson and Robertson, 2012; Sommerville and Sawyer, 1997]. This distinction is not only prevalent in research, but it also influences how requirements are elicited, documented, and validated in practice [Ameller et al., 2012; Borg et al., 2003; Chung and Nixon, 1995; Svensson et al., 2009]. As a matter of fact, up until now there does not exist a commonly accepted approach for the QR-specific elicitation, documentation, and analysis [Borg et al., 2003; Svensson et al., 2009]; QRs are usually described vaguely [Ameller et al., 2012; Borg et al., 2003], remain often not quantified [Svensson et al., 2009], and as a result remain difficult to analyze and test [Ameller et al., 2012; Borg et al., 2003; Svensson et al., 2009]. Furthermore, QRs are often retrofitted in the development process or pursued in parallel with, but separately from, functional requirements [Chung and Nixon, 1995] and, thus, are implicitly managed with little or no consequence analysis [Svensson et al., 2009]. This limited focus on QRs can result in the long run in high maintenance costs [Svensson et al., 2009].

Although the importance of QRs for software and systems development is widely accepted, the discourse about how to handle QRs is still dominated by how to differentiate

them exactly from FRs [Broy, 2016; Glinz, 2007]. One point of view is that the distinction is an artificial one and we should rather differentiate between *behavior* (e.g., response times) and *representation* (e.g., programming languages). The underlying argument is that most QRs actually describe behavioral properties [Glinz, 2007] and should be treated the same way as FRs in the software development process [Broy, 2015]. Behavioral properties subsume traditional FRs, such as *"the user must be able to remove articles from the shopping basket"* as well as QRs which describe behavior such as *"the system must react on every input within 10ms"*. Representational properties include QRs that determine how a system shall be syntactically or technically represented, such as *"the software must be implemented in the programming language Java"* [Broy, 2015, 2016].

In this chapter, we empirically investigate this point of view and aim to increase our understanding on the nature of QRs addressing system properties. To this end, we classify 530 QRs extracted from 11 industrial requirements specifications with respect to their kind. Our results show that 75% of the requirements labeled as "quality requirement" in the considered industrial specifications describe system behavior and only 25% describe the representation of the system. As behavior has many facets, we further classify behavioral QRs according to the *system view* they address (interface, architecture, or state), and the *behavior theory* used to express them (syntactic, logical, probabilistic, or timed) [Broy, 2015, 2016]. Based on this fine-grained classification, we discuss the implications we see on handling QRs in the software engineering disciplines, e.g., testing or design.

Based on the results of our study, we conclude that most "quality requirements" are misleadingly declared as such because they actually describe behavior of the system. This in turn means that many QRs can be handled similarly to functional requirements.

Note. *Please note that the focus of this chapter is not to criticize the term "quality requirement" but to expose the artificial separation of functional and quality requirements in practice.*

5.2. Background & Related Work

In this section, we provide background and related work on requirements categorizations and on the implications of QRs on software development.

5.2.1. Previously Published Material

In our previously published paper [Eckhardt et al., 2015], we presented a research proposal with the goal of analyzing natural language QRs taken from industrial requirements specifications to better understand their nature. Our study reported here, relies on and extends our previous study design. We present the results in full detail, and provide a comprehensive discussion on the implications on software engineering disciplines.

5.2.2. Requirements Categorizations

There are many classification schemes for QRs in literature (e.g., [Chung and do Prado Leite, 2009; Glinz, 2007; ISO/IEC 9126-2001, 2001; Pohl, 2010; Sommerville and Sawyer, 1997; Wagner et al., 2012]). One example for such a classification, which is based on a quality model, is the ISO/IEC 9126-2001 [2001]. It defines external and internal quality of a software system and derives several quality characteristics (e.g., *Functionality–Security* or *Portability–Installability*). Sommerville and Sawyer [1997] further provide a classification scheme based on a distinction between *process requirements*, *product requirements*, and *external requirements*. We base our distinction of QR classes on the ISO/IEC 9126-2001 [2001] classification. Furthermore, we exclude process requirements from our study, as they do not describe properties of the system itself.

Pohl [2010] discusses the misleading use of the term "non-functional" and argues to use "quality requirements" for product-related QR that are not constraints. Glinz [2007] performs a comprehensive review on the existing definitions of QRs, analyzes problems with these definitions, and proposes a definition on his own. He highlights three different problems with the current definitions: a definition problem, i.e., QR definitions have discrepancies in the used terminology and concepts, a classification problem, i.e., the definitions provide very different sub-classifications of QRs, and finally a representation problem, i.e., the notion of QRs is representation-dependent. In our study, we faced all of the three problems: we motivate our study based on the definition and classification problem and during the execution of our study, we faced the representation problem (see also our discussion on threats to validity in Section 5.5). Although we agree on the critique about the obsolete and misleading notion of the term "quality requirement", it still dominates the way requirements are handled in practice, as reflected in our data.

Mairiza et al. [2010] perform a literature review on QRs, investigating the notion of QRs in the software engineering literature to increase the understanding of this complex and multifaceted phenomenon. Amongst others, they found about 114 different QR classes. As a result of a frequency analysis, they found that the five most frequently mentioned QR classes in literature are *performance*, *reliability*, *usability*, *security*, and *maintainability* (in that order). In our study, we got similar results: we found that the five most frequently used QR classes in our industrial specifications are *security*, *reliability*, *usability*, *efficiency*, and *portability* (in that order).[9] While Mairiza et al. [2010] performed their analysis on available literature, our study analyzes QRs documented in industrial projects.

5.2.3. QRs and their Implications on Software Development

One of the first studies that analyzed how to systematically deal with QRs in software development was conducted by Chung and Nixon [1995]. They argue that QRs are often retrofitted in the development process or pursued in parallel with, but separately from, functional design and that an ad-hoc development process often makes it hard to detect defects early. They perform three experimental studies on how well a given

[9] We excluded *functionality* from this list, as it is not a traditional QR class.

framework [Mylopoulos et al., 1992] can be used to systematically deal with QRs. Svensson et al. [2009] perform an interview study on how QRs are used in practice. Based on their interviews, they found that there is no QR-specific elicitation, documentation, and analysis, that QRs are often not quantified and, thus, difficult to test, and that there is only an implicit management of QRs with little or no consequence analysis. Furthermore, they found that at the project level, QRs are not taken into consideration during product planning (and are thereby not included as hard requirements in the projects) and they conclude that the realization of QRs is a reactive rather than proactive effort. Borg et al. [2003] analyze via interviews how QRs are handled in practice by the example of two Swedish software development organizations. They found that QRs are difficult to elicit because of a focus on FRs, they are often described vaguely, are often not sufficiently considered and prioritized, and they are sometimes even ignored. Furthermore, they state that most QR classes are difficult to test properly due to their nature, and when expressed in non-measurable terms, testing is time-consuming or even impossible. Ameller et al. [2012] perform an empirical study based on interviews around the question *How do software architects deal with QRs in practice?* They found that QRs were not often documented, and even when documented, the documentation was not always precise and usually became desynchronized. Furthermore, they state that QRs were claimed to be mostly satisfied at the end of the project although just a few classes were validated. With respect to model-driven development, Ameller et al. [2010] show that most model-driven development (MDD) approaches focus only on functional requirements and do not integrate QRs into the MDD process. They further identify challenges to overcome in order to integrate QRs in the MDD process effectively. Their challenges include *modeling of QRs at the PIM-level*, which includes the question *which QR classes are most relevant to the MDD process?* According to Ameller et al. [2010], the few MDD approaches that support the modeling of QRs can be classified into approaches that use UML extensions [Fatwanto and Boughton, 2008; Wada et al., 2010; Zhu and Liu, 2009] or a specific metamodel [Gönczy et al., 2009; Kugele et al., 2008; Molina and Toval, 2009] to model QRs. In all of the approaches, functional requirements and QRs are modeled separately. Damm et al. [2005] suggest to overcome this separation and propose a so-called rich component model based on UML that integrates functional and QRs in a common model. Similar approaches exist for specific classes of QRs (e.g., for availability [Junker and Neubeck, 2012]). The results of our study provide empirical support for the claim that QRs and FRs are not very different with respect to behavior characteristics and, therefore, can be integrated in a common system model.

All these studies highlight, so far, that QRs are not integrated in the software development process and furthermore that several problems are evident with QRs. In this chapter, we use these problems as motivation and analyze what QR classes can be found in practice and discuss how they can be integrated in the software development process.

5.3. Study Design

In this section, we describe our overall goal, our research questions, and the design of our study.

5.3.1. Goal and Research Questions

The goal of this study is to increase our understanding on the nature of QRs addressing system properties[10]. In particular, we are interested in understanding to which extent these QRs and their respective classes (e.g., security or reliability) describe system behavior and what kind of behavior they address. This allows us to discuss the implications on handling QRs in the software engineering disciplines (e.g., testing or design).

To achieve our goal, we formulate the following research questions (RQs), which we cluster in two categories:

Distribution of QR classes in practice

We examine the distribution of QR classes in practice via two research questions:

RQ1: What QR classes are documented in practice? With this RQ, we want to get an overview of the QR classes that are documented in practice.

RQ2: What QR classes are documented in different application domains? Under this RQ, we analyze whether there is an observable difference between the application domains w.r.t. the documented QR classes.

Nature of the QR classes

We analyze the QR classes with respect to their nature (behavioral or representational) and their kind of behavior via three research questions:

RQ3: How many QRs describe system behavior? With this RQ, we want to better understand how many QRs describe system behavior and how many describe the representation of a system (*behavioral* vs. *representational*) and whether this varies for different QR classes.

RQ4: Which *system views* do behavioral QRs address? With this RQ, we want to better understand the relation between QR classes and the system (modeling) views that the QRs address, e.g., *interface*, *architecture*, or *state behavior*.

RQ5: In which type of *behavior theory* are behavioral QRs expressed? With this RQ, we want to better understand the relation between QR classes and behavior theories used to express the QRs, e.g., logical, timed, or probabilistic description.

[10] In our study, we exclude those QRs going beyond system properties, e.g., process requirements.

5.3.2. Study Object

The study objects used to answer our research questions constitute 11 industrial specifications from 5 different companies for different application domains and of different sizes with 346 QRs[11] in total. We collected all those requirements that were explicitly labeled as "non-functional", "quality", or any specific quality attribute. The specifications further differ in the level of abstraction, detail, and completeness. We cannot give detailed information about the individual QRs or the projects. Yet, in Table 5.1, we summarize our study objects, their application domain, and show exemplary (anonymized) QRs as far as possible within the limits of existing non-disclosure agreements.

5.3.3. Data Collection and Analysis Procedures

To answer our research questions, we prepared the QRs from our study object and then performed a classification and analysis. The procedure was performed by two researchers in a pair. Both have over three years of experience in requirements engineering research and model-based development research.

Data Preparation

The QRs from our study objects differ in their level of abstraction, detail, and completeness. Therefore, we went through the set of QRs and processed each of them in either one of the following ways:

- **Full interpretation:** We considered the QR as it is.
- **Sub interpretation:** We considered only a part of the QR that we clearly identified as desired system property and disregarded the rest of the QR (e.g., due to unnecessary/misleading information).
- **Split requirement:** We split the QR into a set of singular QRs because the original QR addressed more than one desired property of a system.
- **Exclude from study:** We excluded the QR if it was not in the scope of our study (e.g., process requirements), or if we were not able to understand the QR due to missing or vague information.

In total, we excluded 56 requirements ($\approx$ 16%) from the study and considered 76 requirements ($\approx$ 22%) only partially. We split 97 requirements ($\approx$ 28%) into an overall of 337 requirements. Together with the 117 requirements ($\approx$ 34%) that we considered as they are, we ended up with a set of 530 requirements that we used for our classification.

[11] In the data preparation phase, we split non-singular QRs into singular QRs. Thus, the final number of analyzed QRs is 530.

Table 5.1.: Overview of the study objects

Spec.	Application Domain[1]	#Reqs	#QRs	%QRs	Exemplary QR (anonymized due to confidentiality)
S1	BIS (Finance)	200	61	30.5%	*The availability shall not be less than [x]%. That is the current value.*
S2	BIS (Automotive)	177	40	22.6%	*An online help function must be available.*
S3	BIS (Finance)	107	5	4.7%	*The maximal number of users that are at the same time active in the system is [x].*
S4	ES/BIS (Travel Mgmt.)	38	14	36.8%	*The [system] must run on [the operating system] OS/2.*
S5	ES/BIS (Travel Mgmt.)	69	16	23.2%	*It must be possible to completely restore a running configuration when the system crashes.*
S6	ES (Railway)	35	14	40.0%	*The delay between passing a [message] and decoding of the first loop message shall be* $\leq$ *[x] seconds.*
S7	ES (Railway)	122	19	15.6%	*The collection, interpretation, accuracy, and allocation of data relating to the railway network shall be undertaken to a quality level commensurate with the SIL [x] allocation to the [system] equipment.*
S8	ES/BIS (Traffic Mgmt.)	554	128	23.1%	*Critical software components shall be restarted upon failure when feasible.*
S9	ES (Railway)	393	12	3.0%	*The [system] will have a Mean Time Between Wrong Side Failure (MTBWSF) greater than [x] h respectively a THR less than [x]/h due to the use of [a specific] platform.*
S10	ES (Railway)	122	31	25.4%	*The [system] system shall handle a maximum of [x] trains per line.*
S11	BIS (Facility Mgmt.)	24	6	25.0%	*The architecture as well as the programming has to guarantee an easy and efficient maintainability.*
Σ 11		Σ 1.841	Σ 346	18.8%	

[1] We distinguish BIS (Business Information Systems), ES (Embedded Systems), and hybrids of both. For reasons of simplicity, we group various domains (e.g. "Finance") according to single family of systems and use the term "application domain" for that classification.

Data Classification

We classified each of the 530 QRs according to the following classification schemes:

Type of QR: We used the quality model for external and internal quality of the ISO/IEC 9126-2001 [2001] to assign a quality characteristic to each QR (*Functionality–Suitability*, *Reliability–Maturity*, ...; see ISO/IEC 9126-2001 [2001] for details). In our study, the ISO/IEC 9126-2001 [2001] quality characteristics represent the QR class we consider.

System view: We based our classification Broy's requirements categorization [Broy, 2015, 2016] to assign a *system view* to each QR. As illustrated in Figure 5.1, structured views partition QRs into *representational* QRs that refer to the way a system is syntactically or technically represented, described, structured, implemented, or executed (e.g., QR of S4, Table 5.1), and *behavioral* QRs that describe behavioral properties of a system. Behavioral QRs are further partitioned into QRs that describe *black-box* behavior at the *interface* of a system (e.g., QR of S10, Table 5.1) and QRs that address a *glass-box* view onto a system describing its *architecture* (e.g., QR of S8, Table 5.1), or its *state* behavior (e.g., QR of S5, Table 5.1).

Behavior theory: Each *behavioral* QR uses a certain *behavior theory* to express the desired properties of the system. We differentiate between the following classes of behavior theories for our classification:

Syntactic The QR is expressed by a syntactic structure on which behavior can be described (e.g., QR of S2, Table 5.1).

Logical The QR is expressed by a set of interaction patterns (e.g., QR of S8, Table 5.1).

Timed The QR is expressed by a set of interaction patterns with relation to time (e.g., QR of S6, Table 5.1).

Probabilistic The QR is expressed by probabilities for a set of interaction patterns (e.g., QR of S1, Table 5.1).

Timed and probabilistic The QR is expressed by probabilities for a set of interaction patterns with relation to time (e.g., QR of S9, Table 5.1).

To assess the feasibility and clarity of this classification scheme, we performed a pre-study on a subset of the QRs (reported in our previously published material [Eckhardt et al., 2015]). One result of this pre-study was a decision tree for the classification of QRs. We created this tree to improve the reproducibility of our classification (Figure 5.1 shows a simplified version of the taxonomy on which the decision tree is based)[12].

[12]The decision tree can be found under:
`http://www4.in.tum.de/~eckharjo/DecisionTree.pdf` or in Appendix A, Figure A.1.

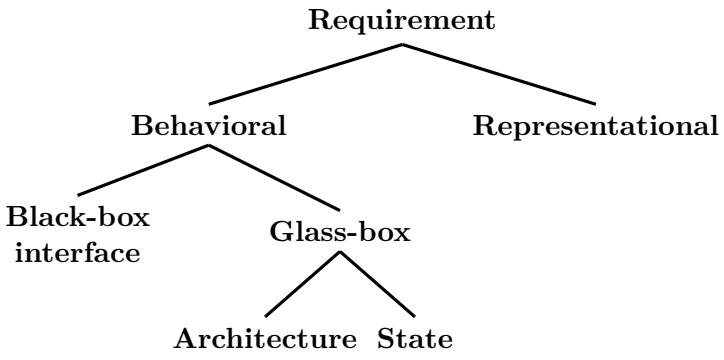


Figure 5.1.: Classification of Requirements by means of the addressed *system view*.

During the pre-study, we also recognized multiple occurrences of QRs following a common *pattern*. For example, many specifications contained an QR following the pattern: *"The system shall run/be installed on platform X"*. We identified a list of 13 of such patterns and assigned a common classification that we applied to all QR instances following that pattern.[13]

Data Analysis Procedures

To answer RQ1, we analyzed the distribution of QRs with respect to the ISO/IEC 9126-2001 [2001] quality characteristics. We provide two views onto this distribution. One detailed view that shows the distribution of QRs with respect to all 27 quality characteristics contained in the standard and one coarse-grained view that shows the distribution of QRs with respect to only 7 aggregated quality characteristics. In the aggregated quality characteristics, we subsumed low-level quality characteristics (such as *Functionality–Suitability* and *Functionality–Accuracy*) to their corresponding high-level quality class (*Functionality* in this case). We made one exception: we created for *Functionality–Security* an own class, as most other QR classifications handle security separately. This results in the following aggregated list of quality characteristics: *Functionality*, *Usability*, *Reliability*, *Security*, *Efficiency*, *Maintainability*, and *Portability* (in the following we will refer to this list as ISO^a quality characteristics).

To answer RQ2, we analyzed the distribution of QRs in the ISO^a quality characteristics with respect to the application domain of the corresponding system.

To answer RQ3, we contrast the number of *representational* QRs with the number of *behavioral* QRs. To answer RQ4, we analyze the distribution of the behavioral QRs with respect to *interface*, *architecture*, and *state behavior*. To answer RQ5, we analyze the distribution of the behavioral QRs with respect to the *behavior theory* used to express them. For each RQ, we present the results for the set of all requirements and structured according to the ISO^a quality characteristics.

[13]The complete list of patterns and the corresponding classification can be found under: `http://www4.in.tum.de/~eckharjo/PatternList.pdf` or in Appendix A, Figure A.2.

Table 5.2.: Distribution of QRs with respect to the ISO/IEC 9126-2001 [2001] quality characteristics

Quality characteristic	**count**	**%**
Functionality - Suitability	117	22.1%
Functionality - Security	104	19.6%
Reliability - Maturity	40	7.5%
Usability - Operability	40	7.5%
Efficiency - Time Behaviour	37	7.0%
Reliability - Reliability Compliance	29	5.5%
Efficiency - Resource Utilization	21	4.0%
Portability - Adaptability	21	4.0%
Portability - Installability	18	3.4%
Maintainability - Changeability	12	2.3%
Reliability - Recoverability	11	2.1%
Functionality - Functionality Compliance	10	1.9%
Usability - Learnability	10	1.9%
Functionality - Accuracy	9	1.7%
Usability - Usability Compliance	9	1.7%
Functionality -Interoperability	8	1.5%
Usability - Understandability	8	1.5%
Maintainability - Analyzability	7	1.3%
Reliability - Fault Tolerance	6	1.1%
Maintainability - Stability	4	0.8%
Portability - Replaceability	4	0.8%
Portability - Co-Existence	3	0.6%
Maintainability - Maintainability Compliance	1	0.2%
Usability - Attractiveness	1	0.2%

5.4. Study Results

In the following, we report on the result for our research questions structured according to the research questions introduced in Section 5.3.

5.4.1. Distribution of QR Classes in Practice

RQ1: Types of QRs

Table 5.2 shows the number (count) and percentage of QRs (relative to the total number of QRs) for each quality characteristic. Table 5.3 further shows the distribution with respect to the ISO[a] quality characteristics. As shown in Table 5.2, the two classes *Functionality–Suitability* and *Functionality–Security* stand out with in total 221 QRs (≈41.7%). *Functionality–Suitability* is defined as *"the capability of the software product to*

Table 5.3.: Distribution of QRs with respect to the ISO[a] quality characteristics.

Quality characteristic	**count**	**%**
Functionality	144	27.2%
Security	104	19.6%
Reliability	86	16.2%
Usability	68	12.8%
Efficiency	58	10.9%
Portability	46	8.7%
Maintainability	24	4.5%

provide an appropriate set of functions for specified tasks and user objectives" [ISO/IEC 9126-2001, 2001]. This essentially corresponds to a traditional understanding of a functional requirement. Furthermore, we classified up to 40 QRs (≈7.5%) as *Reliability–Maturity*, *Usability–Operability*, or as *Efficiency–Time Behaviour*.

In the aggregated results shown in Table 5.3, one can see that the most common classification of QRs is *Functionality* with around 27%. Furthermore, around 20% of all QRs concern *Security*, 16% concern *Reliability*, and 13% concern *Usability*. *Efficiency* (≈ 11%), *Portability* (≈ 9%), and *Maintainability* (≈ 5%) occur only to a small extent in our data.

RQ2: Relation to Application Domain

The results for RQ2 are given in Figure 5.2 showing the distribution of QR quality characteristics with respect to the application domain of the corresponding system (*Business Information System* (BIS), *Hybrid* (ES/BIS), or *Embedded System* (ES)).

One can see a clear difference in the distribution of quality characteristics among the application domains. For example, for business information systems, we classified most QRs as *Security* or *Functionality*, while for embedded systems, most QRs are classified as *Reliability*. In hybrid systems, the distribution among the quality characteristics is more balanced compared with the other application domains.

Although we expected to see different distributions of QR classes between application domains, we were surprised by the extent of this difference. We see these results as a strong argument for domain-specific handling of QRs. In Section 5.6, we will discuss this in more detail.

5.4.2. Nature of Types of QRs

RQ3: Amount of QRs Describing System Behavior

The results for RQ3 are shown in Figure 5.3. The table shows the distribution of *behavioral* and *representational* QRs for all QRs from our data set while the bar chart shows the distribution with respect to the ISO[a] quality characteristics. More precisely, the bar

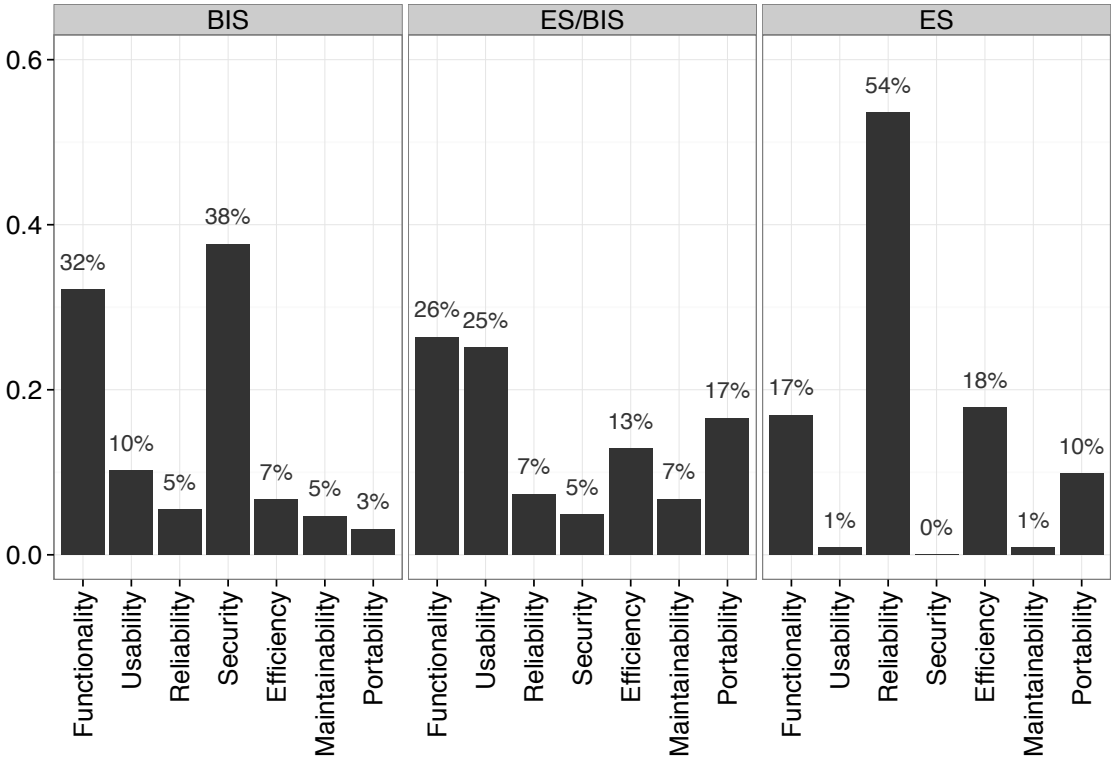


Figure 5.2.: Relative distribution of QRs over the ISO[a] quality characteristics w.r.t. the application domain.

Behavioral vs. Representational	**count**	**%**
Behavioral	396	74.7%
– Black-box	273	51.5%
– Glass-box	123	23.2%
Representational	134	25.3%

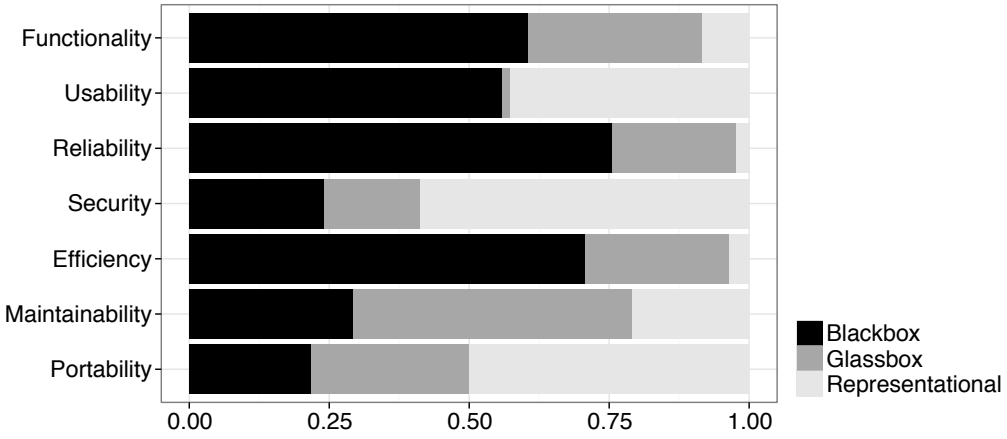


Figure 5.3.: Distribution of *behavioral* and *representational* QRs: *black-box* (black), *glass-box* (dark gray), and *representational* (light gray).

chart shows the percentage of QRs that we classified as *black-box* (black), *glass-box* (dark gray), or *representational* (light gray) within each ISO[a] class.

Quantitative results of RQ3:
74.7% of all QRs describe behavior of the system (black-box or glass-box) while 25.3% describe representational aspects.

More than half of each of the QRs in the *Functionality*, *Usability*, *Reliability*, and *Efficiency* classes describe *black-box* behavior defined over the interface of the system. For example, most efficiency requirements describe desired or expected time intervals between events that are observable at the system interface. Reliability requirements often describe the observable reaction of the system at the interface if an error occurs within the system, such as *"The [system] must have a mean time between failures greater than [x] h"*.

The only class where the largest share of QRs is classified as *glass-box* behavior is *Maintainability*. That means, maintainability requirements, if they consider system properties, often describe the desired internal structure or behavior within this structure (glass-box), as for example the requirement *"The configuration [of the system] shall be independent from the system software and application software"*. However, a substantial amount of *glass-box* behavior can also be found in the *Functionality*, *Reliability*, *Security*,

Efficiency, and *Portability* classes. Thus, QRs within these classes also describe internal behavior, as for example the *Portability* requirement *"The server software shall have the capability to run together with other applications on the same hardware whenever possible"*.

Considering the amount of *representational* QRs, one can see that the QR classes *Usability*, *Portability*, and *Security* stand out. For usability and portability, this is as we expected. Usability requirements often describe representational aspects of the user interface with the goal to support the user in understanding and controlling a system, as for example the requirement *"[The] GUI shall provide a common look and feel whenever possible"*. Portability requirements demand the system to be represented in a way that it fits a specified environment, as for example the requirement *"The system shall run on platform X"*. However, for security, we did not expect such a high portion of *representational* QRs. Therefore, we analyzed these in detail and found that many representational QRs in the *Security* class contain a reference to a standard. For example, we found a high number of QRs stating, *"The security class of the interface to system X with respect to data confidentiality is **high**"*. Excluding those QRs that reference standards from the results, around 54% of security QRs describe *black-box* behavior, 39% describe *glass-box* behavior, and only 7% describe *representational* aspects. This shows that some aspects of security are visible at the interface, as for example user authentication, and some aspects are internal to the system, as for example an encrypted communication within sub-systems.

Another point interesting to us was that none of the QR classes is exclusively *black-box*, *glass-box*, or *representational*. For example, in the class *Functionality*, most QRs describe *black-box* behavior. However, around 31% of the QRs describe glass-box behavior and 17% describe representational aspects. This is because the class *Functionality* does not only include behavior over the interface, but also internal behavior like *"A system component shall save a user's edits whenever possible"*, and also representational aspects like *"The backup data must be stored according to [the company's] policies"*.

RQ4: Distribution of Behavioral QRs w.r.t. System Views

The results for RQ4 are shown in Figure 5.4. The table shows the distribution of QRs with respect to the *system view* they address while the bar chart shows this distribution with respect to the ISO[a] quality characteristics. More precisely, the bar chart shows the percentage of QRs that we classified as *interface* (black), *architecture* (dark gray), or *state* (light gray). For RQ4, we considered only *behavioral* QRs and neglected QRs classified as *representational*, as they do not describe behavior.

Quantitative results of RQ4:
68.9% of all behavioral QRs describe behavior over the interface of the system, 21.5% describe architectural behavior, and 9.6% describe behavior related to states of the system.

System view	count	%
Interface	273	68.9%
Architecture	85	21.5%
State	38	9.6%

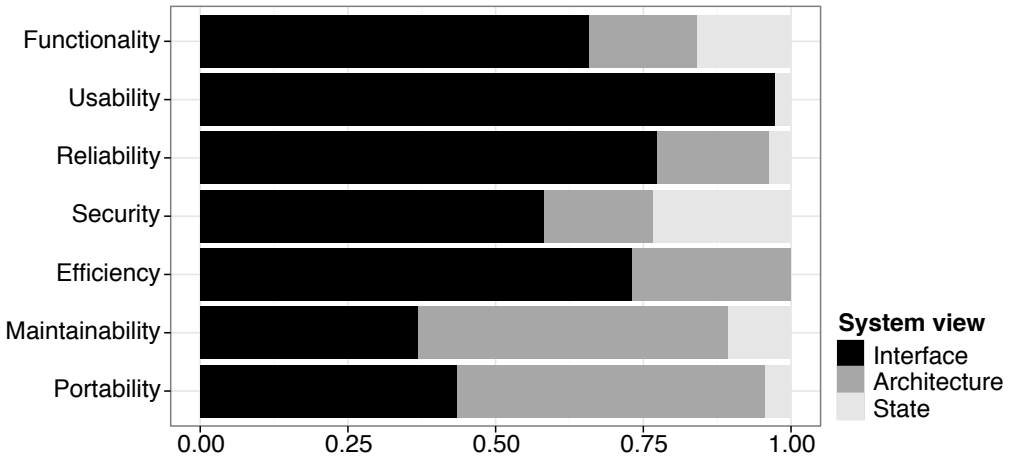


Figure 5.4.: Distribution of *behavioral* QRs with respect to the *system view* they address: *interface* (black), *architecture* (dark gray), and *state* (light gray).

We can see that in the *Functionality*, *Usability*, *Reliability*, *Security*, and *Efficiency* classes most behavioral QRs are classified as *interface*. For the *Maintainability* and *Portability* classes, the most common classification is *architecture*. *Usability* is the only QR class without any QR classified as *architecture*. We can further see that all QR classes but *Efficiency* contain QRs that describe state-related aspects, as for example the *Functionality* requirement *"[The system] must ensure that submitted offers can neither be modified nor deleted"*. This shows that behavioral QRs describe externally visible behavior but also behavior concerning the architecture (see structuring the functionality by functions [Broy, 2010b]) or state-related behavior (see operational states of a system [Vogelsang et al., 2015]). For example, in the *Security* class, there are QRs that describe behavior over the interface like *"There has to be an authentication mechanism"*, some QRs describe architectural behavior like *"[The system] must provide intrusion detection mechanisms"*, and some describe state-related aspects like *"The password shall be valid for at most 30 days"*.

RQ5: Distribution of Behavioral QRs w.r.t. Behavior Theories

The results for RQ5 are shown in Figure 5.5. The table shows the distribution of QRs with respect to the *behavior theory* they use and the bar chart shows this distribution with respect to the ISO[a] quality characteristics. More precisely, the figure shows the percentage of QRs that we classified as *syntactic*, *logical*, *timed*, *probabilistic*, or *probabilistic and*

Behavior theory	count	%
Syntactic	47	11.9%
Logical	277	69.9%
Timed	54	13.6%
Probabilistic	7	1.8%
Probabilistic & Timed	11	2.8%

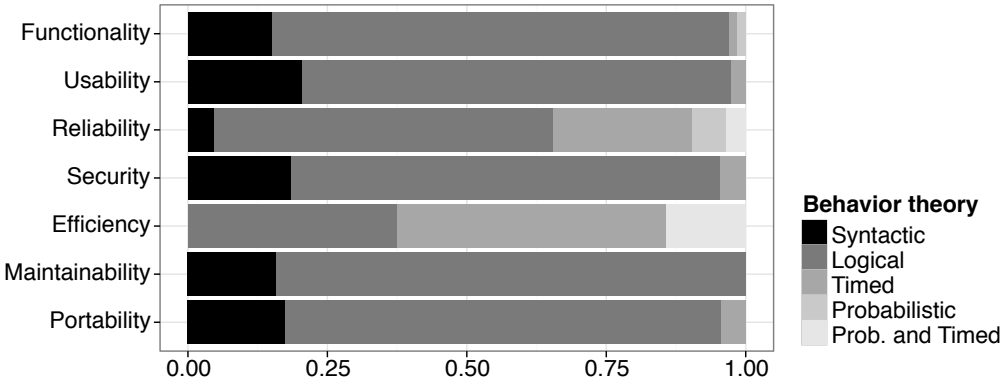


Figure 5.5.: Relative distribution of behavioral QRs with respect to their *behavior theory*: *syntactic*, *logical*, *timed*, *probabilistic*, and *probabilistic and timed* (from black to white).

timed (from black to white). For RQ5, we considered only *behavioral* QRs and neglected QRs classified as *representational* as they do not describe behavior.

Quantitative results of RQ5:
Most behavioral QRs are logical (69.9%), 18.2% are timed and/or probabilistic, and only 11.9% are syntactic.

Over all QR classes, most QRs are *logical* (around 69.9%), while 13.6% are *timed*, 11.9% are *syntactic*, 2.8% are *probabilistic and timed*, and 1.8% are *probabilistic*. Most *timed* and also *probabilistic and timed* QRs belong to the class *Efficiency*. Moreover, the class *Reliability* stands out, as it also contains many *timed*, *probabilistic*, and *timed and probabilistic* QRs.

5.4.3. Summary of Results

Figure 5.6 provides a consolidated quantified view on our overall results.

It shows the distribution of QRs among the QR classes with respect to the addressed *system view* and the used *behavior theory*. The figure shows a table with one diagram per cell; the rows display the QR classes and the columns display the addressed *system view*

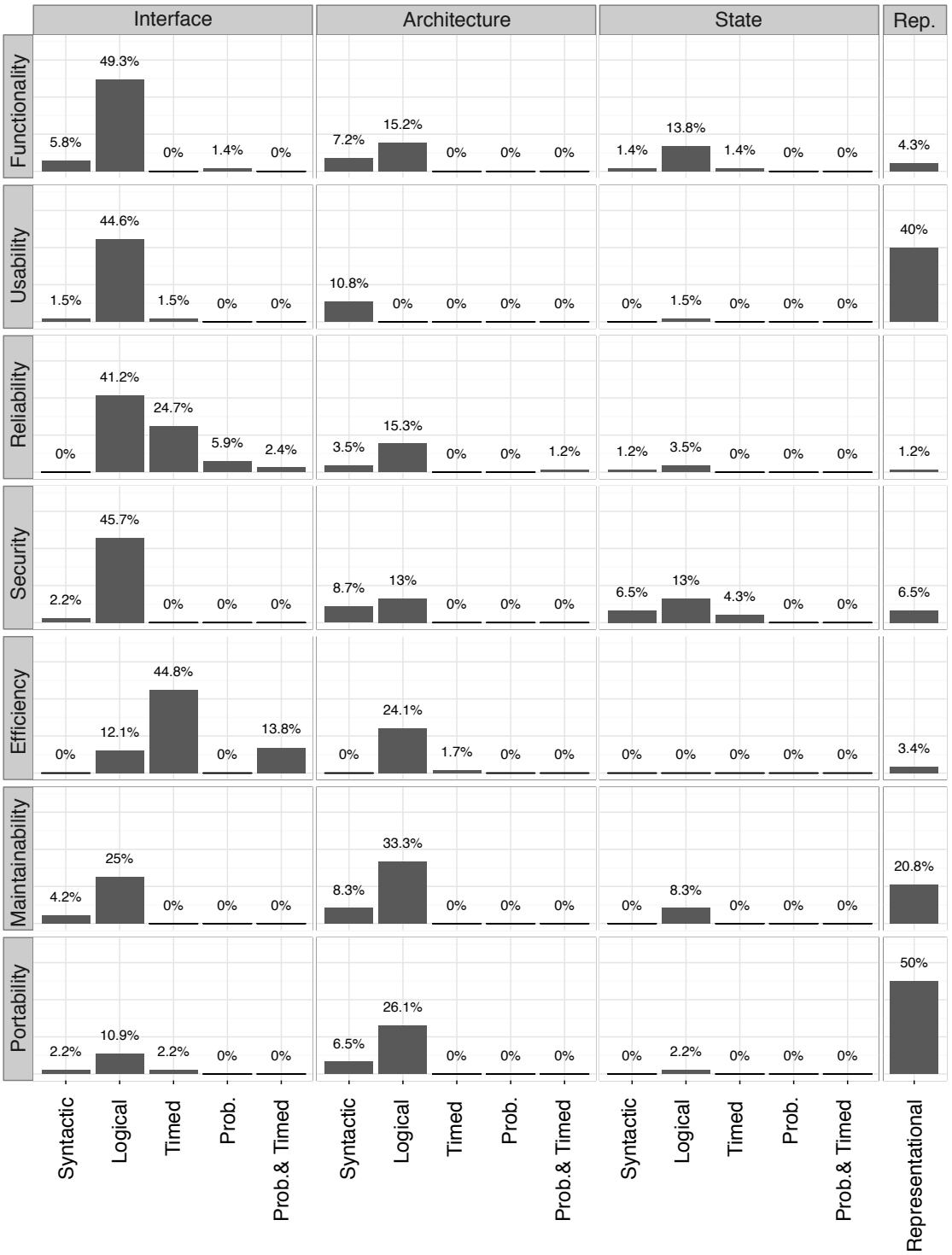


Figure 5.6.: Relative distribution of QRs within the QR classes w.r.t. the addressed *system view* and the used *behavior theory*.

and an additional column for the *representational* QRs. Within each cell, the relative distribution per *behavior theory* is shown (relative per QR class, i.e., the values in all cells of each row sum up to 100%).

In conclusion, most QRs address interface behavior, mostly expressed by logical or timed assertions. The QR classes *Usability*, *Security*, and *Portability* include, in contrast to the other classes, a high portion of *representational* QRs. Furthermore, all classes but *Usability* contain architectural aspects (see column *Architecture*), while the highest percentage of those QRs are in the *Maintainability* class.

5.5. Threats to Validity

In the following, we discuss the threats to validity and mitigation measures we took. We discuss them along the different problems as they arose during our study.

5.5.1. Representativeness of the Data

Inherent to the nature of our study is the data representativeness on which we built our analysis. The concerns range from the representativeness of the way the QRs are specified to the completeness of the data as it currently covers only the particularities of selected industrial contexts. We cannot mitigate this threat but consider our data set large enough to allow us to draw first conclusions on the state of the practice. The relation to existing evidence (see Section 5.7.1) additionally strengthens our conclusions.

5.5.2. QR Selection Problem

We collected only those requirements explicitly labeled as "non-functional" or "quality". With this selection procedure, some relevant QRs might have been missed or irrelevant ones might have been included. To address this problem, we plan to perform the classification on the whole data set as future work, including functional and quality requirements.

5.5.3. Preparation Problem

In our data preparation phase, we excluded QR from the study if they were not in scope of our study (e.g., process requirements) or if we were not able to understand them (due to missing or vague information). This exclusion process could threaten the overall conclusion validity, but as we excluded only about 16%, we do not consider this as a major threat.

5.5.4. Classification Problem

Prior to our study, we performed a pre-study with several independent classification rounds [Eckhardt et al., 2015]. The inter-rater agreement between the independent raters was, however, so low that we had to conclude that the classification dimensions are not clear enough. To resolve this issue, we performed several refinements of the

classification and created a decision tree and a pattern catalogue that supports the classification process [Eckhardt et al., 2015]. In the end, we did the classification in a pair of researchers and individually discussed each QR.

5.5.5. Representation Problem

Although classifying in a pair of researchers, we still faced the *representation problem* discussed by Glinz [2007], which threatens the internal validity. If an QR stated *"The system shall authenticate the user"*, we classified it as *black-box interface*, and *logical* as it describes a black-box behavior over the interface. However, if an QR stated *"The system shall contain an authentication component"*, we classified it as *glass-box architecture* and *logical* as it requires an *internal* component for authentication.

5.5.6. Contextualization Problem

We consider the reliability of our conclusions to be very much dependent on the possibility to reproduce the results, which in turn is dependent on the clearness of the context information. The latter, however, is strongly limited by NDAs that too often prevent providing full disclosure of the contexts and even the project characteristics. To mitigate this threat, we anonymized the data as much as possible and disclosed all information possible within the limits of our non-disclosure agreements.

5.6. Discussion

Based on the results, we identified a set of insights which we discuss in the following paragraphs.

5.6.1. QRs are Not Non-functional

It is commonly acknowledged that functional requirements describe logical behavior over the interface of the system. From a broader view, one could even say that functional requirements describe any kind of behavior over the interface of the system, including timing and/or probabilistic behavior. From this perspective, we conclude that many of those QRs that address system properties describe the same type of behavior as functional requirements do (see column *Interface* in Figure 5.6). This is true for almost all QR classes we analyzed; even for QR classes which are sometimes called *internal* quality attributes (e.g., portability or maintainability) [McConnell, 2004]. Hence, we argue that—at least based on our data—most "quality" requirements describe functional aspects of a system and are, thus, basically *not* non-functional. From a practical point of view, this means that most QRs can be elicited, specified, and analyzed like functional requirements. For example, QRs classified as black-box interface requirements, are candidates for system tests. In our data set, system test cases could have been specified for almost 51.5% of the QRs.

5.6.2. Functional Requirements are often Labeled as QRs

Moreover, functional requirements in the traditional understanding were often labeled as QRs in our examined specifications. We classified 22.1% of our overall QR population as *Functionality–Suitability*, which is a quality characteristic that addresses the functionality of a system (*"The capability of the software product to provide an appropriate set of functions for specified tasks and user objectives"* [ISO/IEC 9126-2001, 2001]). Given that QRs are usually not tested and analyzed as thoroughly as functional requirements [Ameller et al., 2012; Borg et al., 2003; Svensson et al., 2009], this means that, one out of five QRs in our data set elude a thorough analysis process just because they are labeled as QRs.

5.6.3. QRs are often Specified by Reference to Standards

As already indicated within our results for RQ3, we realized that several examined QRs describe requirements by pointing to a standard (e.g., company style guides or safety standards). More specifically, 68 of 530 QRs ($\approx 13\%$) contained references to standards. We classified these as *representational* since we were not able to access these standards due to availability and time constraints. However, these standard-referencing QRs might be interesting to explore in future investigations. On the one hand, they allow a concise specification; on the other hand, they introduce much implicitly necessary knowledge and assume that the reader of the specification has knowledge about and access to those standards.

5.6.4. Only few QRs Deal with Architectural Aspects

While in literature the relation of QRs to architecture and architectural constraints of a system is often emphasized [Chung et al., 2012; Pohl, 2010; Zhu and Gorton, 2007], the QRs of our sample dealt with architecture only to a small degree (see column *Architecture* in Figure 5.6). Only for *Efficiency*, *Maintainability*, and *Portability*, roughly one quarter of the QRs considered architectural aspects of a system. Following this, we argue that—at least based on our data—only few QRs actually describe architectural aspects of a system. It is an interesting point for future research to mirror our findings with the notion of *architecturally significant requirements (ASRs)* [Chen et al., 2013]. ASRs are those requirements which have a measurable impact on a software system's architecture. They are often difficult to define and articulate, tend to be expressed vaguely, are often initially neglected, tend to be hidden within other requirements, and are subjective, variable, and situational [Chen et al., 2013]. Certainly, all QRs that we classified as addressing the system view *architecture* can be considered as QRs. However, also QRs that we classified as addressing the system view *interface* or *state* may have an impact on the architecture, as for example the requirement *"the system should provide five nines (99.999 percent) availability"*. The difference is that QRs addressing the system view *architecture* make the impact on the architecture explicit. For other QRs, an architect needs to decide whether they are ASRs or not.

5.6.5. No QR Class is Uniquely Affiliated with only one Behavior Characteristic

Our analysis shows that none of the considered QR classes is characterized by only one specific *system view* or *behavior theory*. Accordingly, most QR classes contain representational and behavioral QRs which address all *system views* and using all *behavior theories*. While a classification of QRs according to quality characteristics may be helpful to express the intent of an QR, the quality characteristic should not determine how an QR is specified, implemented, or tested. This decision should rather be made based on the addressed *system view* and the *behavior theory* used to express the QR.

5.6.6. The Application Domain Influences QRs

As our results indicate, the application domain of the corresponding system influences the relevancy of QR classes. We therefore conclude that specification and analysis procedures should be customized for different application domains. For example, in the embedded systems domain, the need for probabilistic analysis techniques is stronger compared with the business information systems domain due to the larger amount of reliability QRs that are often described in a probabilistic manner. On the other hand, for business information systems, we should support specification techniques that integrate functional requirements and behavioral QRs, since around 70% of the QRs from BISs were classified as functionality or security (excluding the QRs referencing standards from the class security, most QRs in the class security concern the interface), which describe logical interface behavior to a large extent.

5.6.7. QRs are Specified at Different Levels of Abstraction

In our data set, we found QRs at different levels of abstraction varying in their degree of detail and completeness. QRs ranged from high-level goal descriptions like *"The availability shall not be less than [x]%"* to very concrete and detailed descriptions of behavior like *"The delay between passing a [message] and decoding the first loop message shall be ≤ [x] seconds"*. This is in tune with the view of Pohl [2010]; He states that "non-functional" requirements are underspecified functional requirements. In a development process, high-level QRs need to be refined to detailed functional requirements. To make this refinement explicit, we need an approach for relating high-level QRs to low-level functional requirements. A first approach in this direction is proposed by Broy in his recent work [Broy, 2016].

5.6.8. Structuring the Specification of QRs

There are different ways of structuring functional requirements in requirements specifications, inter alia, structuring the requirements according to functions of the system [Broy, 2010b] or structuring requirements according to components of the system. However, with respect to quality requirements, the question how to structure them in a requirements specification is still open. For example, given the performance requirement *"[The function*

X] must have an average processing time of less than 10ms". This requirement can be structurally added as a child to the requirements of *"function X"*. But the security requirement *"[The system] must ensure that submitted offers can neither be modified nor deleted"* rather belongs to the system as a whole or to a specific component.

Our results of RQ3 and RQ4 show that 74.7% of all requirements describe behavior of the system (black-box or glass-box) and 25.3% describe representational aspects of a system. From the behavioral requirement, 68.9% describe behavior over the interface of the system, 21.5% describe architectural behavior, and 9.6% describe behavior related to states of the system.

If we interpret these results with respect to how to structure QRs in a requirements specification, we could structure all those requirements that are categorized as *black-box interface* requirements into a hierarchy of features and describe each feature by specifications of functions (functional requirement as well as so-called QRs). The remaining requirements, i.e., those that are categorized as glass-box architecture and glass-box state, can be structured into a hierarchy of sub-systems forming its components.

5.7. Conclusions

The goal of this chapter was to reach our second objective. In particular, our goal was to assess if a categorization based on a system model is adequate for requirements found in practice and whether it effectively supports subsequent development activities. In this dissertation, with *adequacy of a requirements categorization*, we mean that the categorization is applicable for industrial requirements and, furthermore, supports subsequent development activities. To this end, we conducted a study where we analyzed and classified 530 QRs extracted from 11 industrial requirements specifications with respect to Broy's requirements categorization that is based on a system model. According to the categorization, 75% of the requirements labeled as "quality" in the considered industrial specifications describe system behavior and 25% describe the representation of the system. From the QRs that describe system behavior, 69% describe behavior over the interface of the system, 21% describe architectural behavior and 10% describe state behavior. We furthermore discussed the implications we see on handling QRs in the software development phases, e.g., testing or design.

Based on these results, we argue that functional requirements describe any kind of behavior over the interface of the system, including timing and/or probabilistic behavior. From this perspective, we conclude that many of those QRs that address system properties describe the same type of behavior as functional requirements do (see column *Interface* in Figure 5.6). This is true for almost all QR classes we analyzed; even for QR classes which are sometimes called *internal* quality attributes (e.g., portability or maintainability) [McConnell, 2004]. Hence, we argue that Broy's requirements categorization—that is based on a system model—is adequate for requirements found in practice, as the categories can be linked to system development activities. From a practical point of view, this means that most QRs can be elicited, specified, and analyzed like functional requirements. For example, QRs classified as black-box interface requirements, are candidates for system

tests. In our data set, system test cases could have been specified for almost 51.5% of the QRs. This contribution supports (the first part of) our hypothesis, i.e., a categorization based on a system model is adequate for requirements found in practice.

In the next chapter, we summarize and critically discuss the implications of requirements categorizations in practice and sketch how to overcome deficiencies associated with QR in practice.

5.7.1. Relation to Existing Evidence

Our results show various relations to existing evidence. For instance, during our classification, we faced all three problems described by Glinz [2007]. We also experienced same or similar terminological confusions on QRs as reported by Ameller et al. [2012]. In particular, we found that categories such as *availability* were often misinterpreted in the documents and used in different ways, e.g., as *performance*. Furthermore, they found that the four QR classes most important to software architects were *performance*, *usability*, *security*, and *availability* (in that order). We could support their results via quantitative results: Their four QR classes are in our list of the top four QR classes (in a different order). Finally, our results also resemble the results of Mairiza et al. [2010] with respect to the five most frequently mentioned QR classes in literature.

Apart from supporting existing evidence, we provide first empirical evidence on what quality requirements are in their nature by analyzing and classifying them with respect to various facets. Summarizing our findings, we conclude that most so-called "quality" requirements in our sample describe functional aspects of a system and are, thus, essentially *not* non-functional.

5.7.2. Impact/Implications

Our results strengthen our confidence that many requirements that are currently classified as QRs in practice can be handled equally to functional requirements, which has both a strong theoretical and practical impact. Existing literature (e.g., [Ameller et al., 2012; Borg et al., 2003; Chung and Nixon, 1995; Svensson et al., 2009]) indicates that the development process for a requirement differs depending on whether it is classified as "quality" or "functional". In contrast to functional requirements, requirements classified as QR are often neglected and properties like testability are not supported. In industrial collaborations, we have also seen that QRs and FRs were documented in separate documents, which has led to failing acceptance tests performed by an external company. Our results suggest that this separation is artificial to a large extent. We argue that treating QRs the same as FRs would have major consequences for the software engineering process. However, there are currently no empirical studies that investigate this argument in detail.

A long-term vision that emerges from our results is that we might be able to better integrate QRs into a holistic software development process in the future. Such an integration would yield, for instance, seamless modeling of all properties associated with a system—no matter if they are functional or quality. The benefits of such an integration

include that QRs would not be neglected during development activities, as it is too often current state of practice; from an improvement in the traceability of requirements over an improvement of possibilities for progress control to an improvement of validation and verification.

5.7.3. Future Work

Our analysis is based on an inherently incomplete set of requirements specifications gathered from practical environments. Hence, our study can be considered as a first attempt to improve the understanding on the nature of QRs from a practical perspective. This has certain implications on the validity of our results (see Section 5.5). However, they still provide a suitable basis to draw first conclusions, which need to be strengthened via additional studies; for instance, by increasing the sample size, by taking into account further application domains, but also by including functional requirements into the analysis. So far, our results are suitable to trigger critical, yet important discussions within the community.

We are planning three concrete next steps based on our data set: First, we will include the remaining 1495 functional requirements (the ones not labeled as "non-functional" or quality attribute) in our study. Second, we are planning to advance the integration of QRs into software development by providing specification blueprints (based on an integrated model) for practitioners. Third, as discussed in Section 5.7.2, we will investigate the consequences of labeling a requirement as "QR" for the development process. We expect to find consequences for how requirements labeled as QR are tested or when they are considered in the development process.

"The most important property of a program is whether it accomplishes the intention of its user."

— Sir Charles Antony Richard Hoare

6 Chapter

An Analysis of Requirements Categorizations and their Consequences in Practice

We gave an overview of requirements categorizations in Chapter 3, analyzed in Chapter 4 how practitioners handle requirements and requirements categorizations, and analyzed the adequacy[1] of a categorization based on a system model with respect to requirements found in practice in Chapter 5. The goal of this chapter is to analyze the results from these chapters and to derive problems with requirements categorizations in practice.

6.1. Deficiencies of Requirements Categorizations in Practice

The results of our study with 109 practitioners described in Chapter 4 indicates that there are several issues with requirements categorizations in practice. Table 6.1 lists the issues (results of a qualitative coding as recommended by (Straussian) Grounded Theory [Stol et al., 2016]) and the number of occurrences of the issues. The table shows the coded answers of the participants that make a distinction between FRs and QRs, i.e., of participants that use a requirements categorization.

If we analyze these issues in detail, we see that there are deficiencies associated with the handling of QRs in all requirements engineering activities. For example, in the *documentation* activity, handling the traceability and the consistency of QRs may

Table 6.1.: Issues and number of occurrence of the code.

Category	**Issue**	#
General	Traceability becomes expensive	4
General	QRs are neglected	2
General	Distinction between QR and FR unclear	2
General	QRs are forgotten	2
General	QR defect may lead to weak user acceptance	2
General	Focus too much on FR	2
General	Maintaining consistency becomes expensive	1
General	QRs are handled differently	1
General	QRs remain high-level	1
General	QRs overlap	1
RE	QRs are collected in a rush	1
Design & Impl.	Late architectural changes	2
V&V	Unclear testing process for QRs	2

become expensive and, furthermore, QRs may remain high-level and may overlap. In the *elicitation* activity, QRs may be neglected or forgotten and the focus may be too much on FR. In the *negotiation* activity, QR defects may lead to weak user acceptance and in the *validation* and *management* activity, QRs may overlap and the traceability may become expensive.

Furthermore, there are several studies in literature that analyze deficiencies that are associated with QRs in a system and software development process. For example, Chung and Nixon [1995] argue that QRs are often retrofitted in the development process or pursued in parallel with, but separately from, functional design and that an ad-hoc development process often makes it hard to detect defects early. Svensson et al. [2009] perform an interview study on how QRs are used in practice and found that there is no QR-specific elicitation, documentation, and analysis, that QRs are often not quantified and, thus, difficult to test, and that there is only an implicit management of QRs with little or no consequence analysis. Furthermore, they found that at the project level, QRs are not taken into consideration during product planning (and are thereby not included as hard requirements in the projects) and they conclude that the realization of QRs is a reactive rather than proactive effort. Borg et al. [2003] found that QRs are difficult to elicit because of a focus on FRs, they are often described vaguely, are often not sufficiently considered and prioritized, and they are sometimes even ignored. Furthermore, they state that most QR classes are difficult to test properly due to their nature, and when expressed in non-measurable terms, testing is time-consuming or even impossible. Ameller et al. [2012] found that QRs were not often documented, and even when documented, the documentation was not always precise and usually became desynchronized. Furthermore, they state that QRs were claimed to be mostly satisfied at the end of the project although just a few classes were validated. Ameller et al. [2010]

show that most model-driven development (MDD) approaches focus only on functional requirements and do not integrate QRs into the MDD process.

In summary, our results and these studies highlight, so far, that requirements categorizations are not (sufficiently) integrated in the software development process and furthermore that several problems are evident with them. We argue that there are two main problems of requirements categorizations with respect to an integration in system development process, i.e., with respect to its operationalization:

1. The definitions of the individual categories of the categorizations are not defined in a way that is easily **understandable** and **clearly applicable** for a given context.

2. The categorizations only provide **minor means to support** subsequent development activities.

Those two problems essentially concern the purpose of a categorization (in general). In our understanding, the purpose of a categorization (in general) is to clearly and unambiguously categorize elements in categories according to clearly defined arguments. Furthermore, and even more important, a categorization should have a clearly defined purpose. For requirements categorizations, this means that on the one hand—from an academic perspective—a categorization should clearly and unambiguously categorize requirements. On the other hand—from a practical perspective—a categorization should categorize requirements in a way such that the activities that are performed with the requirements can be aligned according to the categories.

Note. *In the following, we will discuss the ISO/IEC 25010-2011 [2011] as a representative for a categorization based on quality attributes. In particular, we discuss the system/software product quality model as shown in Figure 3.4.*

6.1.1. Understandability and Separability

The quality model of the ISO/IEC 25010-2011 [2011] consists of in total seven characteristics with 31 sub-characteristics. For each of the characteristics and sub-characteristics a short definition is prose is provided. If we consider the individual definitions in detail, we argue that the definitions are not really helpful to clearly understand the definition and apply them in a given context. For example, the ISO/IEC 25010-2011 [2011] provides the following definition for the quality characteristic "Performance/efficiency":

> *"Performance/efficiency is the performance relative to the amount of resources used under stated conditions. Note: Resources can include other software products, the software and hardware configuration of the system, and materials (e.g. print paper, storage media)."*
>
> — ISO/IEC 25010-2011 [2011]

In the definition, *performance/efficiency* is defined by using the term *performance*. First, the term *performance* is ambiguous as it has many different definitions depending on the context and the ISO/IEC 25010-2011 [2011] does not clarify what meaning they

use for their definitions. In this case, we understand the term *performance* as *the degree to which the system fulfills its purpose*. Still, the definition does not provide a precise notion of the concept under definition. The same holds for the sub-characteristics. For example, the ISO/IEC 25010-2011 [2011] defines the sub-characteristic "Time-behavior" as

> *"The degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements."*
>
> — ISO/IEC 25010-2011 [2011]

The definition contains the concepts *response time*, *processing time*, and *throughput rates* but does not detail or define their actual meaning. Furthermore, the quality sub-characteristic is defined as *"the degree to meet the requirements"*. This in itself is a questionable definition.

In summary, we argue that these definitions lack a precise and explicit definition of the used terms and thus hamper understandability and applicability for a given context. This lack of a precise and explicit definition may have the following consequences in practice:

Requirements may be categorized in a wrong category. An imprecise definition of the categories of a categorization may lead to a wrong categorization of a requirement. If we assume that the development process differs for different categories, as for example, for safety requirements, the ISO/IEC 26262-2011 [2011] standard provides a specific development process and further processual requirements. However, if we wrongly categorize a requirement as safety requirement, we may conduct a labor-intensive development process that may not be necessary. If we categorize a safety requirement in another category, we do not conduct the specific development process and thus do not comply with the standard.

Thus, we argue that an imprecise definition of the categories of a categorization may lead to wrong categorization of requirements, and this, in turn, may have negative consequences if the development process differs for requirements of different categories.

Important information regarding a requirement may be missed. An imprecise definition of the categories of a categorization may lead to incomplete specifications of requirements (incomplete in the sense of (in)completeness a single requirement, i.e., the degree to which a requirement contains all necessary information [Glinz, 2014]). For example, if we consider the definition of "Time-behavior" in the ISO/IEC 25010-2011 [2011], i.e., "The degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements". This definition does not explicitly state what information is required in the specification of a requirement. Thus, it is unclear to the requirements engineer what information to document, and thus, important information may be missing in the specification of that requirement.

Thus, we argue that an imprecise definition of the categories of a categorization may lead to incomplete requirements, and this, in turn, is named as the most frequent cause for project failure in a survey with 58 requirements engineers by Méndez Fernández and Wagner [2013].

6.1.2. Support for Development Activities

Furthermore, the ISO/IEC 25010-2011 [2011] claims that their models are useful for the activities *specifying requirements*, *establishing measures* and *performing quality evaluations*. The authors argue that the defined quality characteristics can be used as a checklist for ensuring a comprehensive treatment of quality requirements, thus providing a basis for estimating the consequent effort and activities that will be needed during system development. However, they do not provide further details for how to accomplish these tasks.

For example, they provide a simple set of questions for four different stakeholders for the activity *specifying requirements*. For "Performance/efficiency" they provide the following questions:

Stakeholder	**Question**
Primary User	How efficient does the user need to be when using the system to perform their task?
Content provider	How efficient does the content provider need to be when updating the system?
Maintainer	How efficient does the person maintaining or porting the system need to be?
Indirect User	How efficient does the person using the output from the system need to be?

These questions can be used as checklist to support the elicitation of requirements. However, they are overly simplistic as they do not provide a means for specifying the requirements. Furthermore, they remain on a high level and thus do not challenge the requirements engineer to think about specific concepts of efficiency (as for example *response times* or *throughput rates*). Furthermore, as the ISO/IEC 25010-2011 [2011] does not provide a means for the specification of requirements, it does not provide support for subsequent development activities like architectural design.

For establishing measures and performing quality evaluations, the ISO/IEC 25010-2011 [2011] provides a reference model for measuring software product quality. Figure 6.1 shows the reference model. Quality properties are measured by applying a measurement method. A measurement method is a logical sequence of operations used to quantify properties with respect to a specified scale. The result of applying a measurement method is called a quality measure element. The quality characteristics and sub-characteristics can be quantified by applying measurement functions. A measurement function is an algorithm used to combine quality measure elements. The result of applying a measurement function is called a software quality measure. In this way software quality

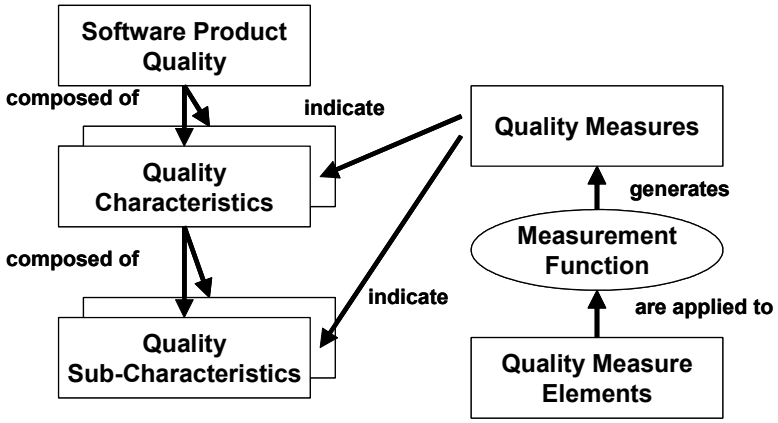


Figure 6.1.: Software product quality measurement reference model (adapted from the ISO/IEC 25010-2011 [2011])

measures become quantifications of the quality characteristics and sub-characteristics. More than one software quality measure may be used to measure a quality characteristic or sub-characteristic.

However, the ISO/IEC 25010-2011 [2011] again remains on a high level and does not provide means to actually implement the measurement reference model in a development process. Moreover, the authors do not consider a specific context in their definition of product quality. In summary, we argue that the ISO/IEC 25010-2011 [2011] provides a comprehensive quality model but remains on a high-level and does not provide guidance for practitioners.

In summary, we argue that a missing means to support for subsequent development activities may have the following consequences in practice:

Ad-hoc handling of requirements Missing means to support and guide subsequent development activities may lead to an ad-hoc handling of requirements of the specific categories. As argued by Chung and Nixon [1995], QRs are often retrofitted in the development process or pursued in parallel with, but separately from, functional design and an ad-hoc development process often makes it hard to detect defects early. For example, in case of missing support for requirements elicitation, the requirements analyst is not guided in asking the right questions. This may lead to incomplete requirements specifications.

Missing acceptance of the Stakeholders Missing means to support and guide subsequent development activities may lead to a missing acceptance of the categorization by stakeholders. As argued before, the purpose of a categorization is to categorize requirements in a way such that the activities that are performed with the requirements can be aligned according to the categories. If this support and guide

is missing, the stakeholders of a categorization, e.g., the requirements engineer may tend to not accept the categorization and thus not use it.

6.1.3. Summary: Deficiencies of Requirements Categorizations in Practice

We consider the ISO/IEC 25010-2011 [2011] a representative for requirements categorizations: As described in Chapter 3.1.2, other requirements categorizations also provide a comprehensive quality model but remain on a high level, as they do not precisely define the individual categories and do not provide support for subsequent development activities. Thus, we follow that current requirements categorizations lack a precise and explicit definition of the individual categories and furthermore do not provide enough guidance for a practitioner to operationalize the categorization for subsequent activities.

6.2. Overcoming the Deficiencies of Requirement Categorizations in Practice

To overcome these two major problems, we propose in the next chapter an approach that clearly and precisely defines the individual quality attributes and, furthermore, provides guidance for practitioners to operationalize the categorization. Given a quality attribute, the core of our approach is two-fold:

1. **Clear definition:** We base the clear and precise definition on the **content elements** that requirements of a specific quality attribute may consist of, i.e., different types of information characterizing the quality attribute (e.g., the desired *latency* of a system for performance requirements). Making the content elements that requirements concerning the quality attribute consist of explicit provides us with a clear vocabulary. For example, instead of defining *Performance/efficiency–Time-behavior* as "The degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements", our approach focusses on the content elements that requirements concerning this quality attribute may consist of. Thus, we provide a content model containing all those content elements the requirement may consist of. This further enables us to provide a definition of the content elements in form of a glossary or mapping to a system model. This reduces the risk of misinterpretations and furthermore facilitates the seamless transition into architectural design.

2. **Sentence patterns:** We base the operationalization on a set of **sentence patterns** for requirements concerning the quality attribute. Sentence patterns provide a way to embody comprehensive and structured knowledge about requirements concerning a specific quality attribute [Withall, 2007]. They support the requirements activities in many ways. For instance, they help the requirements analyst to ask the right questions and to document relevant information on the appropriate level of detail. Moreover, they provide guidance as people tend to learn from examples

> and sentence patterns are abstract examples for the specification of requirements of a specific quality attribute. Furthermore, they give an overview of information that may be documented regarding a specific quality attribute and thus support the elicitation of requirements and—when looking at the sentence patterns as a whole—they support the completeness of the requirements specification as a whole (for a specific quality attribute). In summary, they help to answer the questions "Where do I start?", "How do I know when I am done?", "How detailed should my requirements be?", "Have I missed any requirements?", and "Have I forgotten any critical information in the requirements I have written?" [Withall, 2007].

In summary, given a quality attribute, we base our approach on the clear and precise definition of content elements that requirements of a specific quality attribute may consist of and on a set of sentence patterns for the specification of requirements. Furthermore, we propose to use the idea of activity-based quality models [Deissenboeck et al., 2007; Femmer et al., 2015] for the customization of these content elements to a given organizational context and sentence patterns for guidance and support for their application in practice. Our approach is conducted in advance for a given set of quality attributes. Each application of our approach results in a precise and explicit definition and customized sentence patterns for requirements concerning this quality attribute. The resulting definitions and sentence patterns can then be integrated in the overall RE process of a company to support the elicitation, documentation, validation, and management of requirements in the given organizational context. The results can then be (re)used, for example, as a company standard to specify and elicit quality requirements in future projects.

In the next chapter, we give a detailed description of the approach for defining, specifying, and integrating quality requirements. Furthermore, we apply our approach to two quality attributes: performance and availability.

"A language must have an interpretation for it to serve as a tool for communication. Those who neglect this and those who dogmatically insist that the study of a language independently of its meaning is the only rigorous procedure, are wrong."

— Paul Rosenbloom, 1950

Chapter 7

An Approach for Defining, Specifying, and Integrating Quality Requirements based on a System Model

---

Parts of this chapter have been previously published in the following publications:

- Eckhardt, J., Vogelsang, A., and Femmer, H. (2016a). An Approach for Creating Sentence Patterns for Quality Requirements. In *Proceedings of the 6th International Workshop on Requirements Patterns (RePa)*, pages 308–315 (full paper, workshop, 8 pages)
- Eckhardt, J., Vogelsang, A., Femmer, H., and Mager, P. (2016b). Challenging Incompleteness of Performance Requirements by Sentence Patterns. In *Proceedings of the 24th International Requirements Engineering Conference (RE)*, pages 46–55 (full paper, research track, 10 pages)

In Chapter 4, we analyzed how practitioners handle requirements; We concluded that requirements categorizations should be based on methodological reasons. Moreover, in Chapter 5, we saw that the categorization of Broy [2015, 2016] is suitable for

requirements found in practice. However, it is still an open question how such a categorization can be integrated in an RE process. In this chapter, we provide an approach for defining, specifying, and integrating quality requirements based on a system model and exemplarily instantiate this approach for performance and availability requirements.

7.1. Context

Although the importance of QRs for software and systems development is widely accepted, up until now, there is no commonly accepted approach for the QR-specific elicitation, documentation, and analysis [Borg et al., 2003; Chung and Nixon, 1995; Svensson et al., 2009]. This lack can result in high maintenance costs in the long run [Svensson et al., 2009]. As already argued in the previous chapters, besides Glinz's definition, classification, and representation problem [Glinz, 2007], there are two further problems with current definitions of quality requirements:

1. the definitions are not overly precise and thus not easily understandable and applicable, and

2. the definitions do not provide guidance or support for their application in a given organizational context.

To tackle these two problems, we propose an approach that—given a quality attribute (e.g., performance or availability) as input—provides a means to precisely specify requirements regarding this quality attribute. Our approach is based on the identification of content elements, i.e., different types of information characterizing the quality attribute (e.g., the desired *latency* of a system for performance requirements). In particular, given a quality attribute, our approach provides

1. a means to precisely and explicitly define content elements that are needed to specify requirements concerning the quality attribute, and

2. a set of sentence patterns for practitioners to specify requirements concerning the quality attribute for a given organizational context.

We achieve the precise and explicit definition by a structured identification of relevant content elements that requirements of a specific quality attribute may consist of. Furthermore, we use the idea of activity-based quality models [Deissenboeck et al., 2007; Femmer et al., 2015] for the customization of these content elements to a given organizational context and sentence patterns for guidance and support for their application in practice.

In this chapter, we contribute a detailed presentation and description of our approach, a discussion of our lessons learnt while instantiating it for performance and availability requirements, and provide guidance for how to apply our approach for further quality attributes.

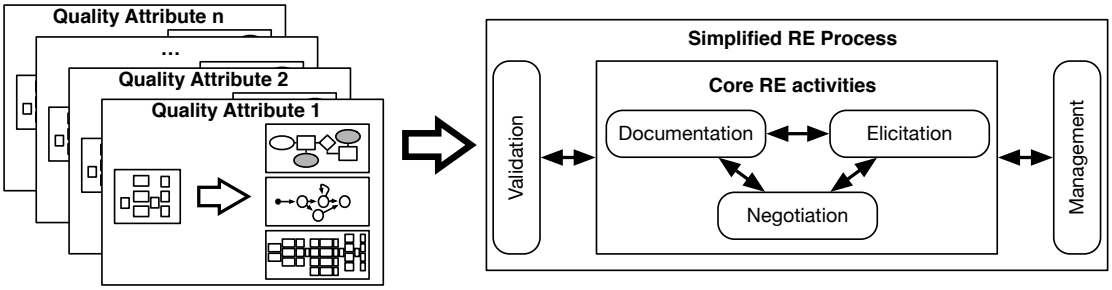


Figure 7.1.: Overview of the approach and integration in a simplified RE process (according to Pohl [2010]). Arrows depict directed relationships.

7.2. Approach & Integration in a RE methodology

Fig. 7.1 shows an overview of our approach and its integration in a RE methodology. On the left-hand side, our approach is shown while on the right-hand side, a simplified RE process (according to Pohl [2010]) is shown. Our approach is conducted in advance for a given set of quality attributes. Each application of our approach takes a specific quality attribute as input and identifies the relevant content elements that requirements of the quality attribute consist of. Based on this, we create a precise and explicit definition and customized sentence patterns for requirements concerning this quality attribute[14]. The resulting definitions and sentence patterns can then be integrated in the overall RE process to support the elicitation, documentation, validation, and management of requirements in the given organizational context. The results can then be (re)used as, for example, a company standard to specify and elicit quality requirements.

7.2.1. Goals of the Approach

Before we describe the approach in detail, we first discuss the goals of the approach. Given a quality attribute we try to achieve the following four goals:

1. **Identification of relevant content elements:** In literature there exists a large amount of publications concerning individual quality attributes. The challenge is to collect this large amount of qualitative data and extract the relevant content elements in a structured and reproducible way that guarantees that all relevant content elements are considered. The quality of the overall results of our approach depends on the content elements that are identified to be *needed to specify requirements concerning the quality attribute.*

2. **Precise definition of relevant content elements:** Given a set of relevant content elements, a further challenge is how to define these precisely such that

[14]The approach provides a means to identify relevant content elements and furthermore to precisely and explicitly define them. However, it is still up to the person conducting the individual steps and thus the quality of the results depends on this.

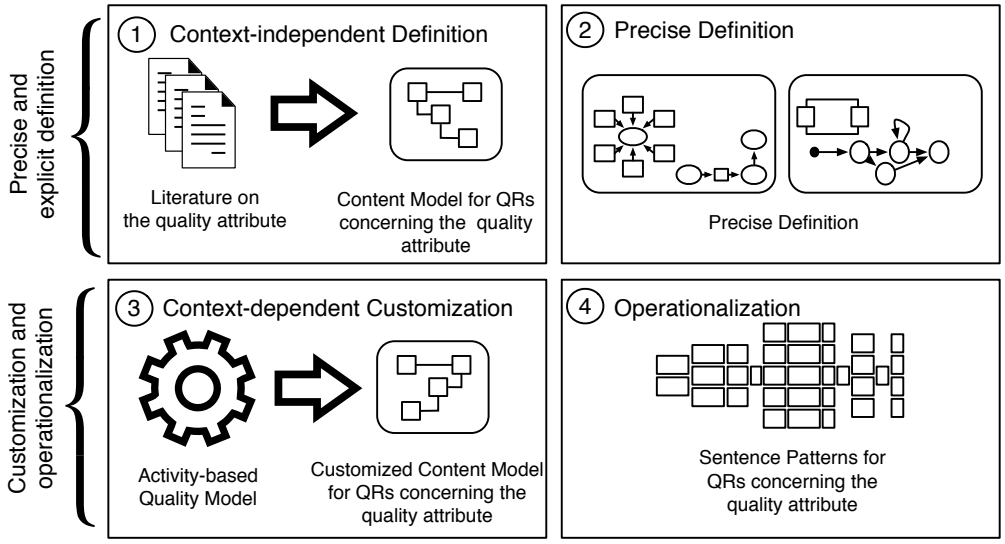


Figure 7.2.: Overview of the approach

all stakeholders have the same understanding. This is a challenging yet creative activity. For example, we may define each content element by means of a glossary entry or give a formal definition by a mapping to a system model. The challenge is to find a way to define the content elements such that they are adequately represented. This activity depends on the context (e.g., involved stakeholders).

3. **Customization to a given organizational context:** Another challenge is to assess whether the content elements are actually relevant for a given organizational context. The simple answer here is to provide a one-size-fits-all solution. However, we argue that such a one-size-fits-all solution does not work for requirements engineering because the organizational contexts vary heavily. These variations include not only the information and level of detail in which projects document requirements but also how projects use requirements documents in their context. Thus, the challenge is to provide an approach that can be customized for a given organizational context.

4. **Provide a means to specify requirements for practitioners:** Based on the relevant content elements, we aim to create a means that supports the structured elicitation, documentation, and management of requirements concerning this quality attribute.

7.2.2. The Approach

Fig. 7.2 shows an overview of our approach. Given a quality attribute, the core of our approach is the structured identification of relevant content elements that requirements of

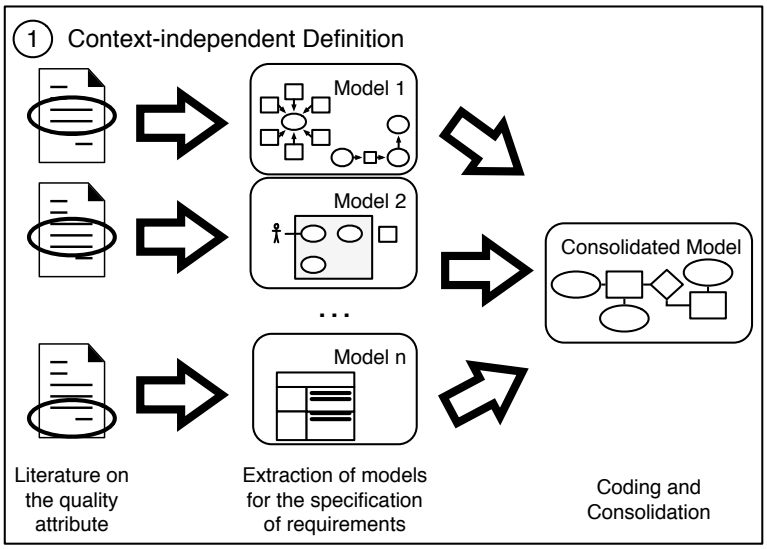


Figure 7.3.: Part 1: Structured identification of relevant content elements.

a specific quality attribute may consist of (Part ①). Based on the resulting content model, our approach provides a precise definition of the content elements (Part ②), a context-dependent customization of the content elements (Part ③), and an operationalization by means of sentence patterns (Part ④). Together, Part ① and ② create a precise and explicit definition of the quality attribute while Part ③ and ④ customize the definition to a specific organizational context and provide a means for practitioners to specify requirements concerning this type. The parts need not necessarily be conducted in that order but Part ① needs to be conducted first, as Part ②-④ are based on it.

1. Context-independent Definition

The result of this part is a comprehensive content model that covers all content elements and relationships that are needed to specify requirements concerning the quality attribute. Fig. 7.3 shows how we propose to structurally identify relevant content elements: To get a complete list of content elements, we propose to use qualitative literature analysis (e.g., a structured literature review or expert interviews) with the goal to identify concepts related to the specification of requirements concerning the quality attribute. Then, in a next step, we identify the models that are used for the specification of requirements; These models can be in textual, semi-formal, or formal form (indicated by different icons in the figure). Then, based on these models, we identify content elements and create a consolidated content model that contains and relates all content elements. This qualitative analysis is a creative and subjective approach. We suggest to use researcher triangulation to reduce this threat to the overall validity.

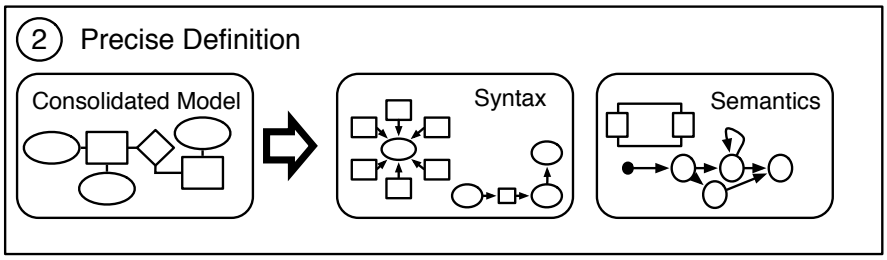


Figure 7.4.: Part 2: Precise definition of the content elements.

2. Precise Definition

Given the content model from the Part ①, the result of this part is a precise definition of the individual content elements of the content model. Fig. 7.4 shows how we propose to reach this precise definition: For each content element of the content model, we define both, its syntax as well as its semantics. Depending on the stakeholders, we may give a definition on different levels of detail ranging from an informal glossary entry to a formal definition. For example, if we chose to use a glossary, we may define the syntax of the content element *scope* (of a requirement), as "The scope of a requirement may be one of *system*, *function*, *component*" and its semantics as "If the scope of a requirement is the *system*, the property described by the requirement must hold for the whole system, if it is a *function*, it must hold for the function, and if it is a *component*, it must hold for the component". If we want to define the content element more formally, we suggest to describe its meaning in terms of a system model (e.g., the system model of the FOCUS theory [Broy and Stølen, 2001]). For example, if we aim to define the semantics of a *requirement*, we can map it to a logical predicate, which relates input streams to output streams. As with Part ①, this part is creative and depending on the context.

3. Context-dependent Customization

The result of this part a customization of the context-independent content model for a given organizational context. To achieve this, we propose to use the idea of activity-based quality models [Deissenboeck et al., 2007; Femmer et al., 2015] and use the context-independent content model as input for the creation of the activity-based quality model for the given context. Fig. 7.5 shows how we propose to conduct this customization. In particular, we use a model of stakeholders and their development activities that take requirements of the quality attribute as input (e.g., *design a test* based on a performance requirement). Based on this model of activities, we successively analyze the content elements that a stakeholder needs in a requirement to complete the activity efficiently and effectively. For example, to perform the activity *designing a test*, it is necessary to know the scope of the requirement. Therefore, we classify content elements as mandatory or optional for an activity. The result of this part is a content model for the quality attribute that is adapted to a specific set of activities and where each content element is

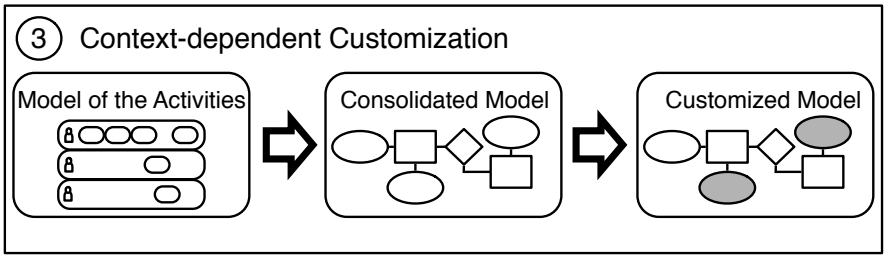


Figure 7.5.: Part 3: Context-dependent customization.

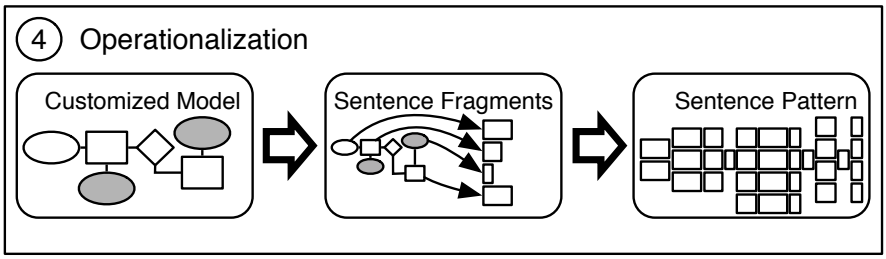


Figure 7.6.: Part: Derivation of sentence patterns.

justified by at least one of these activities. By this, we achieve a customization of the content model to a given organizational context.

4. Operationalization

The result of this part is an operationalization for practitioners, i.e., a means to specify requirements concerning the quality attribute. To achieve this, we propose to derive a set of sentence patterns from the context-dependent content model. Sentence patterns have the advantage that they are easy to use for the documentation of requirements and support the structured elicitation and management of requirements. Fig. 7.6 shows how we propose to derive sentence patterns from a content model. In particular, for each of the content elements in the content model, we derive a *sentence fragment*. The sentence fragment is intended to represent the meaning of the content element as close as possible. For example, let us assume that the content element *scope* of a requirement may be the *system*, a specific *function* X, or a specific *component* C. In this case, we can create the sentence fragments `system` for system, `function <X>` for function, and `component <C>` for component. Furthermore, sentence fragments may also contain variables that have to be replaced by values when the pattern is instantiate. For example, if we aim to represent an optional content element which describes a specific start event, we can represent it by the sentence fragment `[start event <A>]`. The angle brackets indicate the variable while the square brackets indicate that this sentence fragment is optional. Finally, we merge these fragments into sentences. The result of this part is a set of sentence patterns for the specification of requirements concerning the quality attribute.

Summary

In summary, given a quality attribute, the approach derives a context-independent content model based on qualitative literature analysis, provides a clear and explicit definition of the individual content elements, performs a customization for a given organizational context, and provides a means for practitioners to specify requirements concerning the quality attribute for a given organizational context.

7.3. Syntactic Analyses: Challenging Incompleteness

One benefit of our approach is that it creates a context-dependent content model for a given quality attribute. The model is context-dependent in the sense that for a given context, the content model contains all necessary information to complete subsequent activities efficiently and effectively. We can now leverage this fact to support syntactic analyses and introduce a notion of (syntactic) completeness for requirements of this type.

In particular, we define the completeness of requirements for a given quality attribute with respect to the presence of all mandatory content elements in the context-dependent content model. We call a requirement complete if all mandatory content elements are present in the textual representation of the requirement. To give an example, let us assume that a performance requirement has a quantifier that describes whether the requirement specifies an exact value (e.g., "the latency shall be 10ms"), a mean or median (e.g., "the latency shall be on average 10ms"), or a minimum or maximum value (e.g., "the latency shall be at maximum 10ms"). Now we can distinguish three cases for the presence of mandatory content in the textual representation of a requirement:

- The requirement **does not contain** the content. For example, in case of a performance requirement stating *"The delay between [event A] and [event B] shall be short"*, the content regarding the quantifier is not contained.

- The requirement **implicitly** contains the content. With implicit, we mean that the content is contained in the requirement, but we need to interpret the requirement to derive the content. For example, in case of a performance requirement stating *"The delay between [event A] and [event B] shall typically be 10ms"*. In this case, regarding the quantifier, we can interpret "typically" as "median".

- The requirement **explicitly** contains the content. With explicit, we mean that the content is contained without interpretation. For example, in case of a performance requirement stating *"The delay between [event A] and [event B] shall have a median value of 10ms"*. In this case, regarding the quantifier, the content is explicitly contained.

We can now derive the following definitions for incompleteness and for weak and strong completeness of requirements of a given quality attribute:

Definition (Incompleteness)**.** *A requirement of a given quality attribute is* ***incomplete****, if at least one mandatory content elements (w.r.t the context-dependent content model of the attribute) is* ***missing*** *in its textual representation.*

Definition (Weak Completeness)**.** *A requirement of a given quality attribute is* ***weakly complete****, if all mandatory content elements (w.r.t the context-dependent content model of the attribute) are* ***explicitly or implicitly*** *contained in its textual representation.*

Definition (Strong Completeness)**.** *A requirement of a given quality attribute is* ***strongly complete****, if all mandatory content elements (w.r.t the context-dependent content model of the attribute) are* ***explicitly*** *contained in its textual representation.*

We argue that this definition of completeness for requirements of a given quality attribute can be used to detect incompleteness and thus to pinpoint to requirements that are hard to comprehend, implement, and test. For example, requirements of class *incomplete* are not testable at all, requirements in class *weakly complete* need to be interpreted by the developer and tester and therefore bear the risk of misinterpretations, and requirements in class *strongly complete* contain all content necessary to be implemented and tested. Thus, we argue that our approach further provides a helpful and actionable definition of completeness for quality requirements. This definition of completeness can then be used to support analytic as well as constructive quality control.

7.4. Application to Performance Requirements

In this section, we give guidance on how the individual parts can be conducted. In particular, we apply our approach to the quality attribute *performance*, or *performance/ efficiency requirements* as they are called in the ISO/IEC 25010-2011 [2011].

Note. *In the following, we explicitly focus on externally visible performance and exclude internal performance (sometimes also called efficiency), which describes the capability of a product to provide performance in relation to the use of internal resources.*

7.4.1. Context-independent Definition

The result of this part is a comprehensive content model that covers all content elements and relationships that we need to specify requirements concerning the quality attribute. In the last section, we proposed to use qualitative literature analysis for this purpose.

We reduced the set of relevant literature to classifications and categorizations of non-functional and quality requirements (and software and systems quality models) [Behkamal et al., 2009; Boehm et al., 1976; Botella et al., 2004; Dromey, 1995; Glinz, 2005, 2007; Grady, 1992; ISO/IEC 25010-2011, 2011; ISO/IEC 9126-2001, 2001; McCall et al., 1977; Robertson and Robertson, 2012; Sommerville, 2007]. Fig. 7.7 gives a high-level overview of the results of the literature review for performance requirements. In particular, literature differentiates three types of performance requirements: *Time behavior* requirements, *Throughput* requirements, and *Capacity* requirements. Time behavior requirements

Performance			
Time Behavior	**Throughput**	**Capacity**	**Aux. Conditions**
E.g., response time, latency	E.g., rate of transactions, data volume per unit of time	E.g., concurrent users, size of database or storage, transaction limits, user limits	- Measurement location - Measurement period - Load - Platform - Scope of measurement

Figure 7.7.: Overview of performance.

specify *fixed time constraints* like *"The operation Y must have an average response time of less than x seconds"*, throughput requirements specify *relative time constraints* like *"The system must have a transaction rate of x transactions/second"*, and capacity requirements specify *limits of the system* like *"The system must support at least x concurrent users"*. Furthermore, literature defines further aspects related to performance requirements that apply for all three types of performance requirements. We call these aspects *auxiliary conditions* (e.g., the location of a measurement). We give a precise definition for these types in Section 7.4.2.

We coded the results of the literature review as suggested by Grounded Theory [Adolph et al., 2011] to assemble a conceptual model of the quality attribute in form of a content model. The resulting content model contains content elements of the quality attribute and relations between them. Furthermore, we added content elements that apply to requirements in general (e.g., the scope of a requirement). The result of this part is a content model for performance that is a superset of all performance aspects mentioned in literature. The resulting content model is shown in Figure 7.8. The model consists of three parts (Part ① - ③ in the figure). In the following, we describe the individual parts of the content model.

① **Content Elements of Performance Requirements:** A *Performance Requirement* is a *Requirement*. A *Performance Requirement* possibly has a *Selection*, e.g., is it valid for all functions or only for a subset of all functions. A *Performance Requirement* has a *Scope*. The *Scope* can be either the *System*, a *Function*, or a *Component*. Finally, a *Performance Requirement* has a *Quantifier*. The *Quantifier* describes whether the requirement specifies an *Exact Value* (e.g., "the latency shall be 10ms"), a *Mean* or *Median* (e.g., "the latency shall be on average 10ms"), or a *Minimum* or *Maximum* value (e.g., "the latency shall be at maximum 10ms").

② **Content Elements Related of the Specific Performance Requirements Types:** A *Performance Requirement* can be a *Time Behavior Requirement*, a *Throughput Requirement*, or a *Capacity Requirement*.

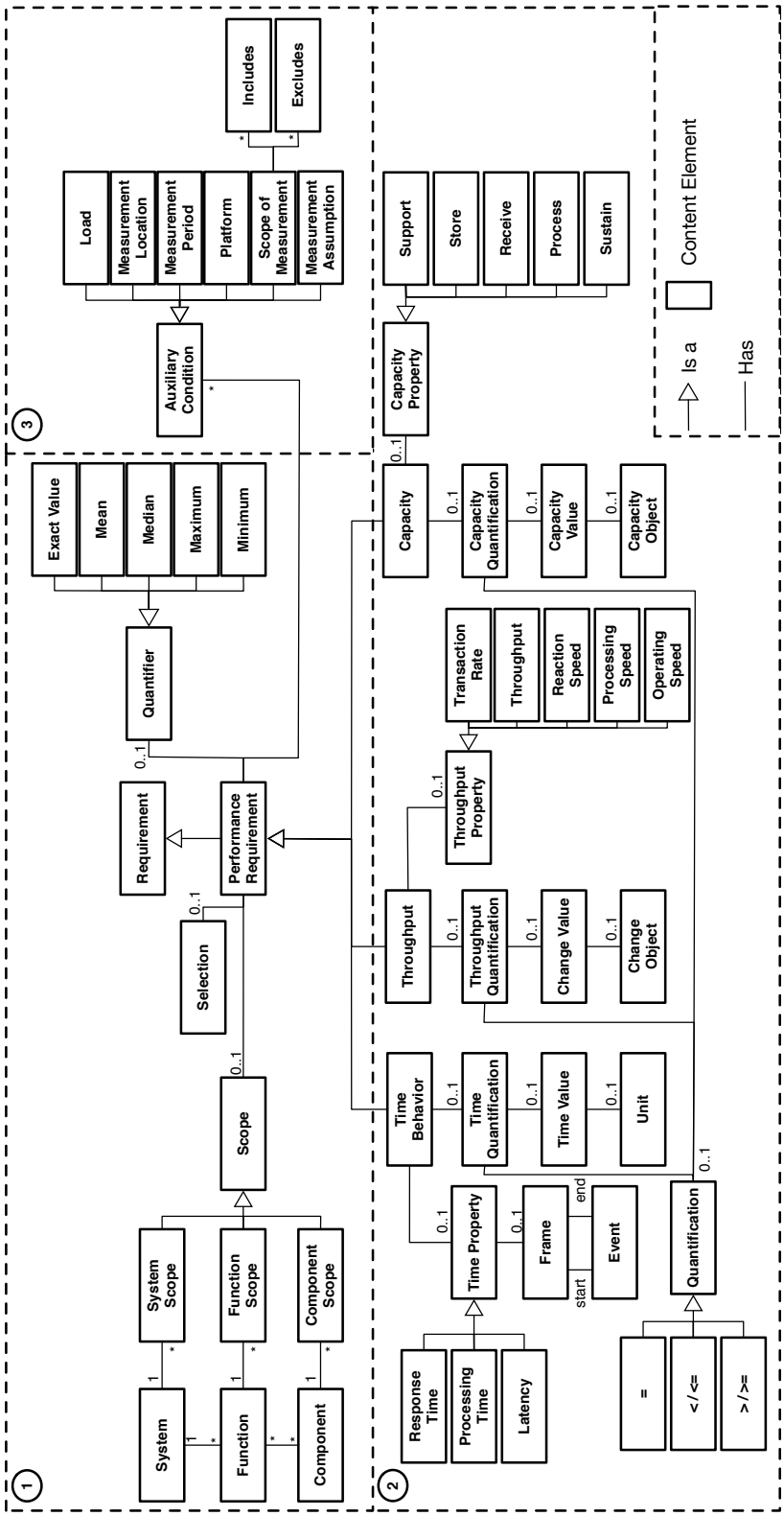


Figure 7.8.: Context-independent content model of performance requirements. It shows content elements of performance requirements and their relationships.

Time Behavior Requirements A *Time Behavior Requirement* describes a *Time Property*. A *Time Property* can be *Response Time*, *Processing Time*, or *Latency*. Furthermore, a *Time Property* may have a *Frame* specifying a *start* and an *end Event* (e.g., "the processing time between event A and event B shall be less than 10ms"). Finally, a time behavior requirement has a time quantification, which quantifies a time value with a specific unit (e.g., "less than 10 ms").

Throughput Requirements A *Throughput Requirement* describes a *Throughput Property*. A *Throughput Property* can be *Transaction Rate*, *Throughput*, *Reaction Speed*, *Processing Speed*, or *Operating Speed*. Finally, a *Throughput Requirement* has a Throughput Quantification, which quantifies a *Change Value* which specifies a *Change Object* per *Time Value* (e.g., "less than 10 users per ms").

Capacity Requirements A *Capacity Requirement* describes a *Capacity Property*. A *Capacity Property* can be *Support*, *Store*, *Receive*, *Process*, or *Sustain*. Finally, a *Capacity Requirement* has a *Capacity Quantification*, which quantifies a *Capacity Object* with respect to a *Change Value* (e.g., "less than 10 concurrent users per 1s").

③ **Content Elements of Auxiliary Conditions:** A *Performance Requirement* may contain (possibly many) auxiliary conditions. An *Auxiliary Condition* may be a specific *Load* (e.g., "at maximal load"), a specific *Measurement Location* (e.g., "in London"), a specific *Measurement Period* (e.g., "between 12/20 and 12/24"), a specific *Platform* (e.g., "on ARMv8"), a specific *Scope of Measurement* specifying the *Includes* and *Excludes* (e.g., "included is the browser render time, but the network time is excluded"), and *Measurement Assumptions* specifying further assumptions for the measurement (e.g., "a specific signal is assumed to be present").

7.4.2. Precise Definition

In this section, we (informally) define the individual types of performance requirements (as shown in Figure 7.7) and discuss how we can express them based on the FOCUS system model.

Running Example We first introduce our running example. It is a simple request-reply system where each message is associated with the user that sent this message. After having received a request by a particular user, the system ignores any further request by this user, before it replies to the request. Thus, for every user, our system satisfies the following two properties:

1. The liveness property, that every request is answered by a response at a *later* point in time.

2. The safety property, that a response is only sent if (one or more) requests have been received and they have not yet been answered.

To formally specify our system, we assume a set of user identifiers U and two special messages *req*, *rep*. To associate the messages with users, we create the sets $I = \{(req, u) \mid u \in U\}$ and $O = \{(rep, u) \mid u \in U\}$, respectively. Then, the syntactic interface of our system S is $(\{i\} \blacktriangleright \{o\})$, with input channel i, output channel o, and $type(i) = I$ and $type(o) = O$.

First, we give some examples of valid and invalid behavior of our system. Let us assume an input stream which contains two requests from user u_1, one at time 0 and one at time 2. Formally, we can represent this stream as follows:

$$\langle (req, u_1), \surd, \surd, (req, u_1) \rangle^\wedge \langle \surd \rangle^\infty$$

Given this input stream, we may have for instance the following two valid output streams:

$$o_1 = \langle \surd, (rep, u_1), \surd, \surd, \surd, \surd, (rep, u_1) \rangle^\wedge \langle \surd \rangle^\infty$$
$$o_2 = \langle \surd, \surd, \surd, \surd, \surd, \surd, (rep, u_1) \rangle^\wedge \langle \surd \rangle^\infty$$

In o_1, each request is answered individually, i.e., at time 1 a reply is sent to user u_1 and at time 5 the second request is answered. In o_2, both requests are answered with only one reply at time 6. Thus, these output streams fulfill our liveness property, i.e., every request is answered by a response at a *later* point in time and the safety property, i.e., a response is only sent if (one or more) requests have been received and they have not yet been answered.

Given our input stream, we may have the following two invalid output streams:

$$o_3 = \langle \surd \rangle^\infty$$
$$o_4 = \langle \surd, (rep, u_1), \surd, (rep, u_1), \surd, \surd, \surd, (rep, u_1) \rangle^\wedge \langle \surd \rangle^\infty$$

In o_3, no request is answered while in o_4, we have three replies: one at time 1, one at time 2, and one at time 5. Both output streams violate our properties: o_3 does not answer any request, thus, the liveness property is violated and o_4 violates the safety property, as the response at time 2 does not answer any open request.

Now we specify our two properties with the predicate $g : \overline{I} \times \overline{O} \to \mathbb{B}$.

$$g(i, o) \Leftrightarrow \forall u \in U :$$
$$(*) \Big(\forall n \in \mathbb{N} \colon i.n = (req, u) \Rightarrow \exists n' \in \mathbb{N} \colon time(o, n') > time(i, n) \wedge o.n' = (rep, u) \Big) \wedge$$
$$(**) \Big(\forall n \in \mathbb{N} \colon o.n = (rep, u) \Rightarrow \exists n' \in \mathbb{N} \colon time(i, n') < time(o, n) \wedge i.n' = (req, u) \wedge$$
$$\Big(\forall n'' < n \colon time(o, n'') > time(i, n') \Rightarrow o.n'' \neq (rep, u) \Big) \Big)$$

In this predicate, equation $(*)$ specifies the liveness property, i.e., every request is answered by a response at a later point in time, and equation $(**)$ specifies the safety property, i.e., a response is only sent if (one or more) requests have been received and

they have not yet been answered. Note that for every $i \in \overline{I}$, it is possible to construct at least one $o \in \overline{O}$ satisfying the two properties. Thus, the predicate induces a total behavior. Now, we can specify the semantic interface of our example system with the following timed FOCUS specification:

Request-Reply	timed
in $i : I$ out $o : O$	
$g(i, o)$	

Event To define the different types of performance requirements, we first introduce the notion of an *event* of a system. An event of a system happens at a specific point in time. With respect to the FOCUS system model, we can define three types of events: (i) an *external interface event* is visible at the external interface of the system, (ii) an *internal architectural event* is visible at the (internal) architecture of a system, and (iii) an *internal state event* is visible at the state view of a system.

An example of an external interface event is the receiving of an input to a specific function at time t_1 (e.g., in our running example, receiving the (req, u) message) and an example of an internal state event is changing the internal state of the system at t_2. We now use the notion of an event to define time behavior and throughput requirements.

Time Behavior Requirements Time behavior requirements are requirements that restrict the time between specific events of the system.

Relation to the Focus system model: Time behavior requirements restrict the time between specific events of the system. Thus, for specifying these requirements, we need to make the events explicit. For example, the (external interface) event that input is sent to a function or the (external interface) event that output is received from a function. The FOCUS theory provides means to specify all kinds of events; external interface events can be specified as predicates over the input and output of a system, internal architectural events can be specified as predicates over the architecture of a system, and internal state events as predicates over the states and state transitions of the system. Thus, the challenging part for time behavior requirements is to identify, specify, and relate these events.

Example (Time behavior requirement in our running example)**.** *An example for a time behavior requirement is the response time of a specific function, i.e., the time between sending the input to the function and receiving the output from the function. For example, w.r.t. our running example, the time behavior requirement "the time between sending a request and receiving the reply shall be less than 100 ticks". It can be formally specified as follows:*

Request-Reply Time behavior — timed

in $i : I$

out $o : O$

$g(i, o) \wedge$

$\forall u \in U, t \in \mathbb{N} : (\exists n \in \mathbb{N} : i.n = (req, u) \wedge time(i, n) = t) \Rightarrow$

$\exists n' \in \mathbb{N}, t' \in]t, t + 100[\ : o.n' = (rep, u) \wedge time(o, n') = t'$

Throughput Requirements Throughput requirements are requirements that restrict the rate of specific events of the system. In contrast to time behavior requirements, which restrict the time between specific events, throughput requirements restrict the rate, i.e., the number of specific events per time frame.

Note (Difference between throughput and time behavior requirements)**.** *Note that it makes a difference if we restrict the rate of specific events vs. the time between the resulting events. For example, consider the throughput requirement "The rate of transaction X shall be more or equal than 10 per second" and the time behavior requirement "The time between transaction X started and ended shall be less or equal than 0.1 seconds". A system fulfilling the time behavior requirement also fulfills the throughput requirement, but a system fulfilling the throughput requirement does not necessarily fulfill the time behavior requirement.*

Relation to the Focus system model: Similar to time behavior requirements, throughput requirements describe a timed relation of events of the system. However, throughput requirements describe the rate, i.e., the number of these events in a given period of time. Thus, following our argumentation for time behavior requirements, the challenging part for throughput requirements is to identify, specify, and relate the events.

Example (Throughput requirement in our running example)**.** *An example for a throughput requirement is the transaction rate of a specific function, i.e., the number of transactions per time frame. For example, w.r.t. our running example, the throughput requirement "The system shall answer at least 10 open requests within 1000 ticks".*

To formally specify this requirement, we first introduce the auxiliary function

$$open : \overline{I} \times \overline{O} \times \mathbb{N} \to \mathbb{N}$$

Given an input stream i, an output stream o, and a point in time t, it returns the number of users that are waiting for a response at time t, i.e., the number of open requests at time t. Formally, it is specified as follows:

$$open(i, o, t) = |\{u \mid \exists n \in \mathbb{N}\colon i.n = (req, u) \wedge time(i, n) \leq t \wedge \\ (\neg\exists n' \in \mathbb{N}\colon o.n' = (rep, u) \wedge time(i, n) < time(o, n') \leq t)\}|$$

Now we can formally specify the requirement:

Request-Reply Throughput — timed

in $i : I$

out $o : O$

$$g(i, o) \wedge \forall t \in \mathbb{N}\colon \underbrace{|\{(u, n) \mid \exists n \in \mathbb{N}\colon o.n = (rep, u) \wedge t < time(o, n) \leq t + 1000\}|}_{\text{The number of replies within }]t, t+1000]} \geq min(10, open(i, o, t))$$

Note (Design Decisions in our Formalization of Throughput Requirements)**.** *In this example, we require that for all times t, the number of replies within $]t, t+1000]$ is greater or equal to the minimum of the number of open requests, and 10. Note that we count each response of each user individually (see (u, n) in the set). Thus, we guarantee that at least 10 replies are sent, given that there are at least 10 open requests. Otherwise, we guarantee that the system responds to at least the number of open requests at time t.*

Moreover, we only count the number of replies and do not identify the individual request-reply pairs. Consequently, we may count different users in time frame $]t, t + 1000]$ who receive a response in contrast to users who have an unanswered request up to time t. However, this fits our understanding of throughput as the rate of successful message delivery over a communication channel [Jansen and Prasad, 1994; Peterson and Davie, 2011; Rappaport, 2002]. Note that, if we want to restrict the duration for a request-reply pair, we can specify this with a time behavior requirement. Throughput requirements mainly consider the output of the system.

Moreover, given that less than 10 users are requesting, we only guarantee that the system sends this number of responses. Thus, we do not require that the system answers all requesting users as soon as possible but the system can delay answering the requests to fulfill the throughput requirement. For example, given that only one user sends a request, the system may wait until the end of the time frame to

answer the request. This still fulfills the throughput requirement. However, with this strategy, we do not require the system to answer the requests as soon as possible and as fast as possible. Still, this fits our understanding of throughput requirements, as we only consider the output rate of the system. We could specify this requirement by combining a throughput requirement with a time behavior requirement.

Capacity Requirements Capacity requirements are requirements that restrict the quantity of a particular entity at specific points in time. An example of a particular entity are concurrent users. It is important to note that capacity requirements restrict behavior at specific *points* in time, which can be either time intervals or the whole runtime of the system.

Relation to the Focus system model: In comparison to time behavior or throughput requirements, which restrict a timed relation between *events* of the system, capacity requirements restrict the quantity of a specific entity. Thus, we first need to make explicit which entity we want to restrict and how we want to measure the quantity. For example, we may want to restrict the number of concurrent users the system shall support. Thus, in this case, we need to find a way to express the number of concurrent users. For concurrent users, there is no explicit notion in the FOCUS system model. However, we still can express users by, for example, tagging each message with an identifier of the user. Thus, for each entity we want to restrict, we have to either find a way to express it based on the FOCUS system model or extend the system model to include such a notion.
Example (Capacity requirement in our running example)**.** *With respect to our running example, an exemplary throughput requirement would be "The system shall guarantee to support up to 1000 concurrent users". Under "support", we understand that the system answers the requests (without restricting the time).*

To formally specify the requirement, we first introduce the auxiliary function violations. Given an input stream i and $c \in \mathbb{N}$, it returns the set of time units in which more than c users send a request in the time frame $[t, t+100[$. Thus, in our example, $violations(i, 1000)$ returns all violations of the precondition of the capacity requirement. The function $violations : \overline{I} \times \mathbb{N} \to \wp(\mathbb{N})$ is specified as follows:

$$violations(i, c) = \{\ t \in \mathbb{N} \mid |\{u \mid \exists n \in \mathbb{N} \colon i.n = (req, u) \wedge t < time(i, n) \leq t + 100\}| > c \}$$

Thus, given an input stream $i \in \overline{I}$, $violations(i, 1000) = \emptyset$ means that the precondition of the capacity requirement is not violated at any time and $violations(i, 1000) \neq \emptyset$ means that there is (at least) one point in time in which the precondition is violated.

Based on this distinction, we can specify the requirement as follows:

Request-Reply Capacity — timed

in $i : I$

out $o : O$

$(violations(i, 1000) = \emptyset \Rightarrow g(i, o)) \wedge$
$(violations(i, 1000) \neq \emptyset \Rightarrow \exists i' \in \overline{I}, o' \in \overline{O} : i' \downarrow_n = i \downarrow_n \wedge o' \downarrow_n = o \downarrow_n \wedge g(i', o'))$

where $n = index(i, min(violations(i, 1000)))$ and the *index* function returns for a given input stream i and time t the least index n such that $time(i, n) = t$.

Note. *We made a distinction based on the precondition* $violations(i, 1000)$ *of the capacity requirement. If there are no violations, i.e.,* $violations(i, 1000) = \emptyset$*, we guarantee that the requests are answered as specified in the relation* g*. If there are violations, i.e.,* $violations(i, 1000) \neq \emptyset$*, we only guarantee up to the first violation that requests are answered as specified in* g*. After that, we do not guarantee anything. Thus, in this case, everything can happen. This fits our understanding of capacity requirements [Jansen and Prasad, 1994]. If the system is for example under a distributed denial of service (DDoS) attack, i.e., a large amount of users are requesting, the system may crash or drop messages.*

Auxiliary Conditions Auxiliary conditions contain further aspects of performance requirements.

Load Load for performance requirements describe profiles of how many users are using the system and how they are using the system.

Example: For example, *high load* represents a specific number of users that perform a specific number of requests per second. An exemplary requirement is *"the system shall have a transaction rate of x per second while under high load."*.

Relation to the Focus system model: Load profiles can be expressed based on the FOCUS system model. As already discussed above, we can include a notion of users in the system model. For specifying the load profiles, the A/C specification style (see e.g. [Broy, 1998]) suits well, as user behavior can be modeled as interaction patterns with the system.

Measurement location The measurement location for performance requirements describes the (physical) location of parts of the system and, furthermore, communication latencies between these.

Example: For example, consider a system that consist of three sub-systems: A, B, C. A may run in Frankfurt, B in Munich, and C in Berlin. The communication latencies may then be 40 ms between Frankfurt and Berlin, 50 ms between Berlin and Munich, and 30ms between Munich and Frankfurt.

Relation to the Focus system model: The measurement location of performance requirements can be expressed in the Focus system model. In particular, a notion of a deployment of components (and also functions) to physical locations and of communication latencies between these can be expressed based on the Focus system model.

Measurement period The measurement period for performance requirement describes the actual time the requirement is valid. It furthermore contains specific load profiles, i.e., at what time is how much load on the system.

Example: For example, the load profile may be given as follows: $[0-8[$ low load, $[8-17[$ high load, $[17-24[$ medium load. An exemplary requirement is *"the system shall have a transaction rate of x per second between 8 o'clock and 17 o'clock"*.

Relation to the Focus system model: Similar to the load of a performance requirement, the measurement period of performance requirements can be expressed based on the Focus system model.

Platform The platform of the measurement specifies a specific platform (together with an operational model) for the system.

Example: An exemplary requirement is *"the system shall have a transaction rate of x per second while running on an ARMv7 processor"*.

Relation to the Focus system model: For expressing the platform of the measurement, we need an operational model. An operational model is not contained in the Focus system model. Thus, to include the platform, we need to extend the system model.

Scope of Measurement The scope of the measurement specifies what sub-systems are included in the scope of the requirement and what sub-systems are excluded.

Relation to the Focus system model: The scope of the measurement can be expressed based on the Focus system model, by including an excluding specific components.

Summary

Time behavior requirements and throughput requirements restrict the time for or the rate of specific events of the system. For both, the challenging part is to identify, specify, and relate these events. External interface events can be specified as predicates over the input and output of a system, internal architectural events can be specified as predicates over the architecture of a system, and internal state events as predicates over the states and state transitions of the system. In contrast to time behavior and throughput requirements, capacity requirements restrict the quantity of a specific entity. Thus, we first need to define which entity we want to restrict and how we want to observe it. Finally, for the

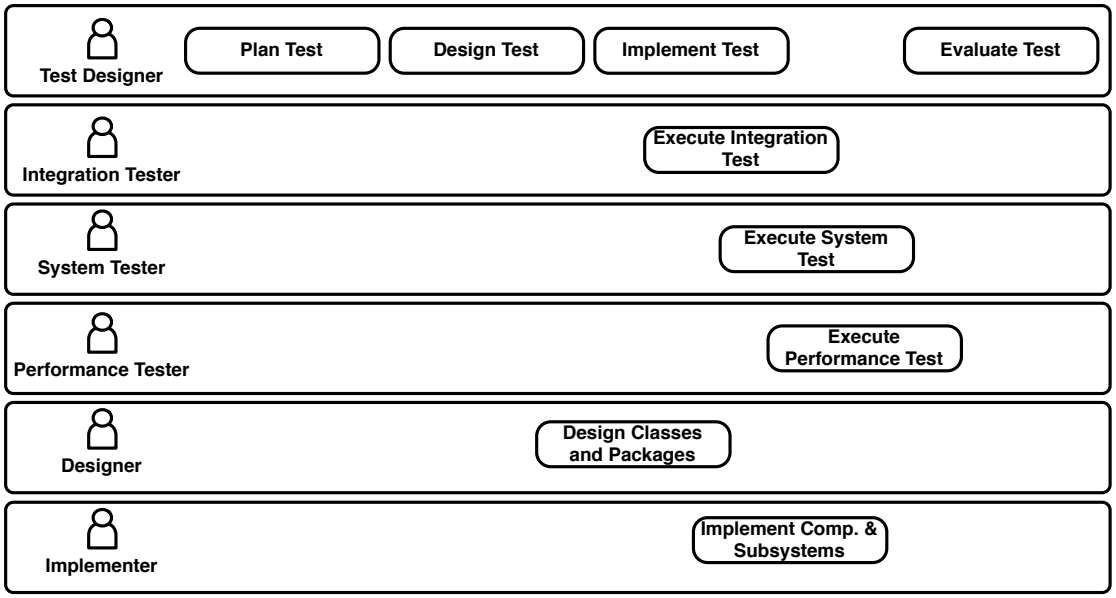


Figure 7.9.: The activities for testing from the rational unified process [Jacobson et al., 1999].

auxiliary conditions, we can express all but the platform of the measurement based on the FOCUS system model. For specifying performance requirements, the A/C specification style (see e.g. [Broy, 1998]) suits well, as these types of requirements often describe an interaction pattern between an user and the system, which can be expressed easily based on A/C.

7.4.3. Context-dependent Customization

The result of this part is a customization of the content model for a given operational context (see Part ③ in Fig. 7.2). To achieve this, we follow the idea of activity-based quality models [Deissenboeck et al., 2007; Femmer et al., 2015] and use the context-independent content model as input for the creation of the activity-based quality model for the given context. Here, we first consider all stakeholders and analyze their development activities that take requirements of the quality attribute as input, such as *design test* of the *test designer* in case of performance requirement. We identify necessary and important content elements that these requirements must contain to complete the development activities efficiently and effectively. We accordingly classify content elements, marking crucial content elements as mandatory and the contributing content elements as optional. The result of this part is a content model that is customized for a given operational context and each content element is justified by at least one development activity.

For performance requirements, we used *testing activities* as described in the (rational) unified process (RUP) [Jacobson et al., 1999]. Figure 7.9 shows the activities and

Table 7.1.: Necessary content elements to complete development activities efficiently and effectively.

Stakeholder	**RUP activities**	**Necessary content element**
Test designer	Plan Test	Scope
	Design Test	Scope, Time Property, Throughput Property, Capacity Property
	Implement Test	Scope, Quantifier, Time Property, Throughput Property, Capacity Property, Time Quantification, Time Value, Unit, Throughput Quantification, Change Value, Change Object, Capacity Quantification, Capacity Value, Capacity Object
	Evaluate Test	Scope, Quantifier, Time Property, Throughput Property, Capacity Property, Time Quantification, Time Value, Unit, Throughput Quantification, Change Value, Change Object, Capacity Quantification, Capacity Value, Capacity Object
System tester	Execute System Test	Scope, Quantifier
Performance tester	Execute Performance Test	Scope, Quantifier
Designer	Design Classes and Packages	Scope, Time Property, Throughput Property, Capacity Property
Implementer	Implement Components and Subsystems	Scope, Time Property, Throughput Property, Capacity Property, Time Quantification, Time Value, Unit, Throughput Quantification, Change Value, Change Object, Capacity Quantification, Capacity Value, Capacity Object

the associated stakeholders. For each of the stakeholders' activities, we identified the corresponding necessary content elements from the content model to complete the activity efficiently and effectively. As the description of the activities in the RUP is rather high-level and does not provide detailed insights about the required artifacts for an activity, we performed an in-depth analysis of the description of the respective activities. Then, in a pair of researchers, we discussed the activities and identified the necessary content elements of a requirement for that activity: We marked a content element as necessary when we agreed that its absence would require a stakeholder to invest additional effort for completing the activity or would even make the activity impossible. Table 7.1 shows the resulting mapping between the necessary content elements and the activities. For example, for the activity *design test* by the *test engineer*, the *time/throughput/capacity property* of the requirement is necessary as its absence would make it impossible to set up an adequate test environment. Furthermore, the scope of the requirement is necessary for the activity *plan test*, as the test engineer needs this information for assigning the test to a person/team responsible. Figure 7.10 shows the final context-dependent content model. We marked a content element as necessary if it is necessary for at least one development activity (marked as gray in the figure). The resulting content model provides a justification for each of the individual content elements of the context-independent content model.

7.4.4. Operationalization

In this section, we provide an operationalization for practitioners based on sentence patterns. In particular, we provide a set of sentence patterns for performance requirements. These sentence patterns are based on the context-dependent contend model for performance requirements (see Figure 7.10) and can be used to specify performance requirements that follow a specific pattern. Given a clear definition of the individual content items, we can now specify performance requirements which are built based on the content model and which can be understood based on the definition.

The result of this part is an operationalization for practitioners, i.e., a means to specify requirement concerning the quality attribute. To achieve this, we propose to derive a set of sentence patterns from the context-dependent content model (see Part ④ in Fig. 7.2). In particular, for each of the content elements in the content model, we derive a *sentence fragment*. The sentence fragment is intended to represent the meaning of the content element as closely as possible. Finally, we merge these fragments into sentences. The result of this part is a set of sentence patterns for the specification of requirements concerning the quality attribute.

We iterated through the set of content items in a pair of researchers and discussed how to adequately represent this content element in terms of a sentence fragment. Figure 7.11 shows the resulting patterns. In order to build a sentence, a requirements engineer must first select the performance requirement type, i.e., one of *Time Behavior*, *Throughput*, or *Capacity*. Then, sentences can be specified from left to right, while selecting one of the sentence fragments and replacing the variables in angle brackets. Sentence fragments in square brackets (e.g., `[between event ⟨A⟩ and event ⟨B⟩]` in Figure 7.11a) are op-

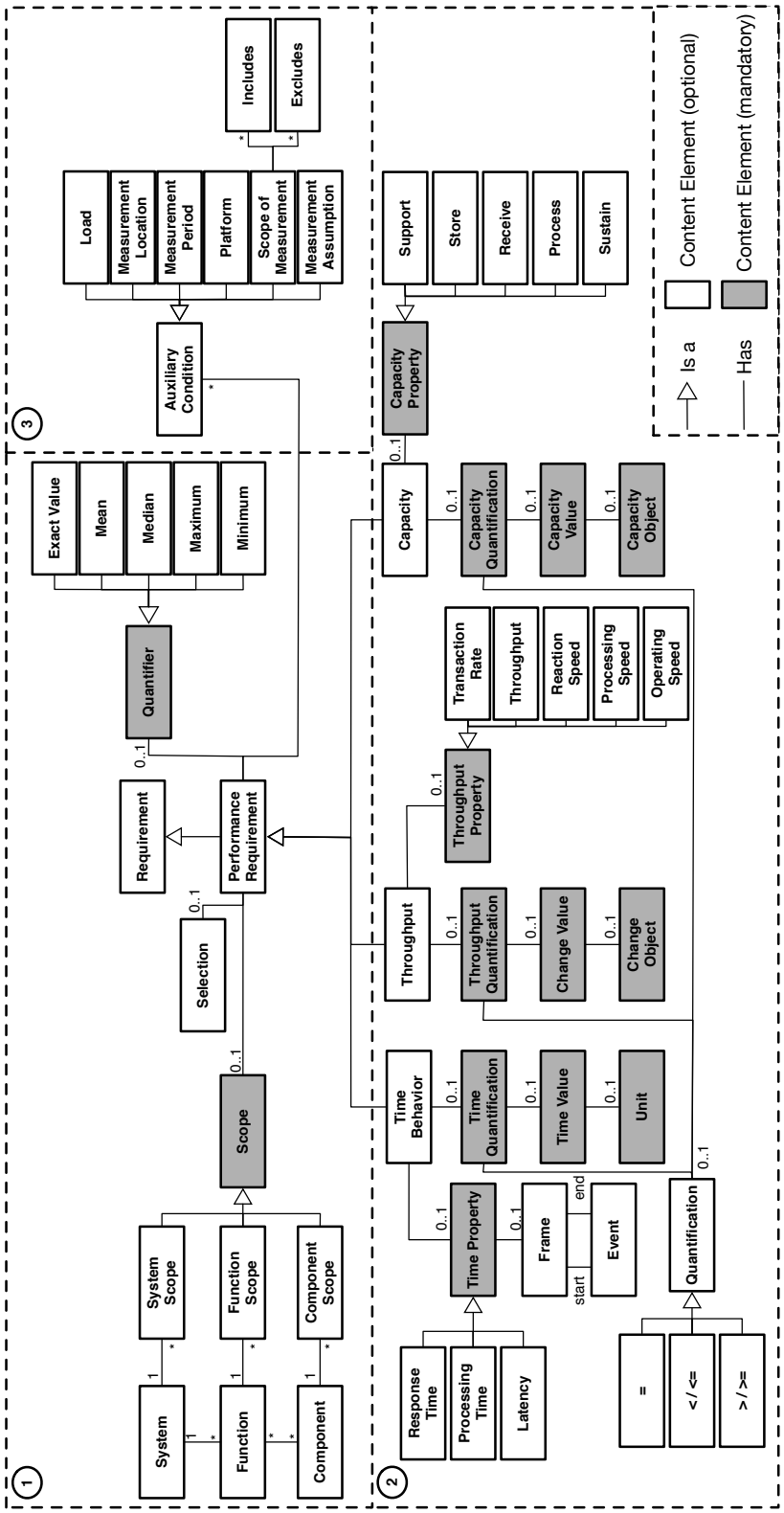


Figure 7.10.: Context-dependent content model for performance requirements. It shows content elements of performance requirements and their relationships. Content elements with a white background depict optional content and with a gray background depict mandatory content with respect to the analyzed activities.

tional. Then, auxiliary conditions can be added by applying the sentence patterns in Figure 7.11d. Exemplary sentences are

> The system must have a processing time of < 10 ms between event "receiving a request" and event "answering a request", when under a maximal load. Measurement takes place on production hardware. Included is browser render time.

or

> The system must be able to process a maximum of 10.000 requests per s. Measurement takes place in Munich, Bavaria. Excluded are external services.

7.5. Application to Availability Requirements

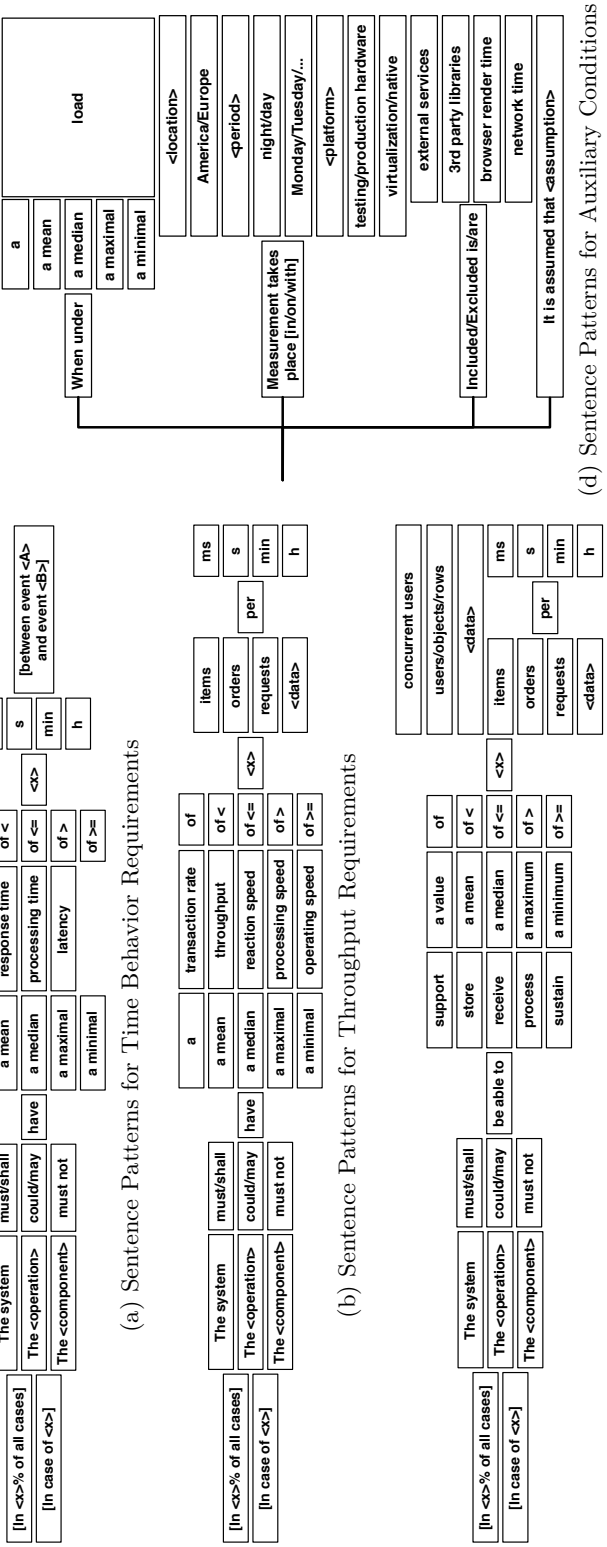
In this section, we apply our approach to the quality attribute *availability*. The results of this section are based on the work of Junker [2016]. In his work, Junker [2016] provides an artifact model and a modeling method for the specification and analysis of availability for software-intensive systems. In this section, we apply our approach to availability and rely on the formalization and artifact model provided in the work of Junker [2016]. In particular, we extend the artifact model by a content model for availability (see Section 7.5.1) and provide sentence patterns for the specification of availability requirements (see Section 7.5.2). The precise definition of the individual elements of the content model can be found in the work of Junker [2016].

7.5.1. Context-independent Definition

The result of this part is a comprehensive content model that covers all content elements and relationships that we need to specify availability requirements. For the context-independent definition, we base our analysis on the Availability Artifact Model of Junker [2016]. Figure 7.12 shows the model; Blue artifacts are based on a generic artifact model whereas green artifacts are the extensions introduced by Junker for availability. In the following, we will focus on the green artifacts:

Availability Requirements Specification The *availability requirements specification* contains the requirements of the system regarding availability. It captures the demands of the different stakeholders of the system that relate to availability. Availability requirements are captured informally with so-called *textual availability descriptions*, i.e., statements about availability formulated in natural language. Availability requirements may refer to functional requirements and to elements of the functional architecture. These informal descriptions are formalized in a so-called *availability constraints model*.

Availability Specification The *availability specification* contains the necessary definitions of failure and of availability metrics. It essentially provides definitions of



(a) Sentence Patterns for Time Behavior Requirements

(b) Sentence Patterns for Throughput Requirements

(c) Sentence Patterns for Capacity Requirements

(d) Sentence Patterns for Auxiliary Conditions

Figure 7.11.: Sentence Patterns for Performance Requirements

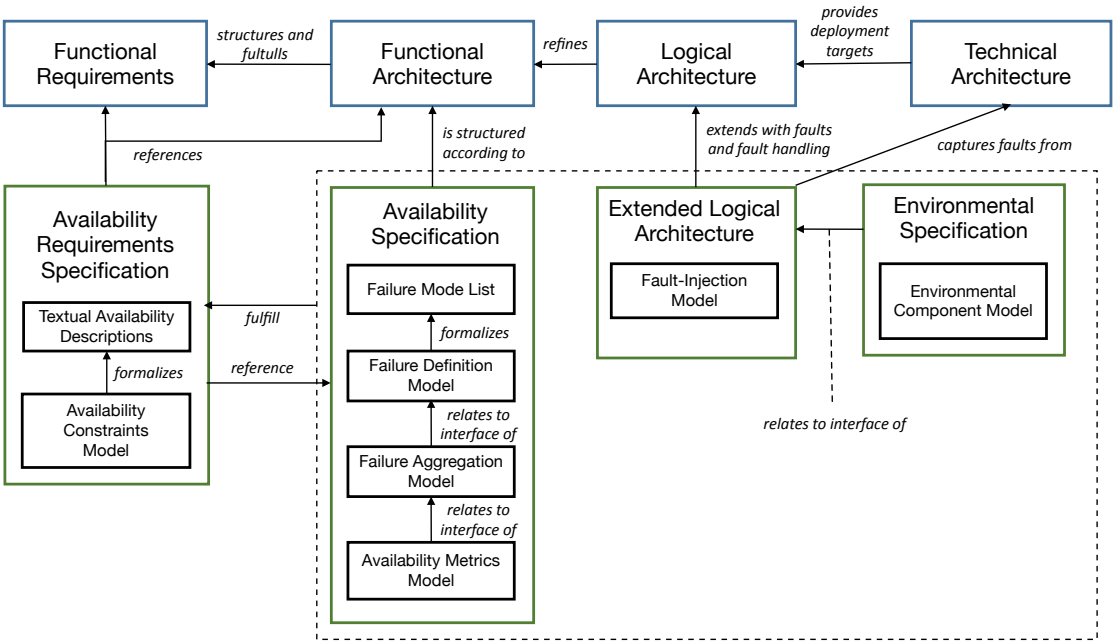


Figure 7.12.: Overview over the Availability Artifact Model from Junker [2016]. Boxes with a black border denote the model types used in these artifacts. Arrows describe the relationship between models and artifacts.

what availability should mean for the given system. In particular, it provides a definition of what failure means and defines how to calculate availability metrics. The *failure mode list* describes informally the failure modes considered relevant to describe the availability of the system. The *failure definition model* formalizes the failure mode list, and, in particular, relates deviations from the specified behavior of the system to the failure modes. The *failure aggregation model* is an optional model that aggregates and simplifies failure modes to facilitate the definition of availability metrics. Finally, the *availability metrics model* defines how to calculate metrics.

Extended Logical Architecture The *extended logical architecture* is an extension of the original logical architecture to include the behavior in case of faults. The *fault-injection model* includes, potentially for every component in the logical architecture, a description of how this component is affected by faults and how these faults lead to a change in behavior.

Environment Specification The *environment specification* describes the behavior and structure of external systems and users. The *environment component model* describes the structure and behavior of the environment.

To create the content model, we iterated through all artifacts of the artifact model in a pair of researchers and extracted the content elements that are needed for the

specification of availability requirements. The resulting content model contains content elements of the quality attribute and relations between them. Furthermore, we added content elements that apply to requirements in general (e.g., the scope of a requirement). The resulting content model is shown in Figure 7.13. The central content elements are the definition of failures, the definition of metrics, and requirements constraining metrics w.r.t. specific failures.

As with performance requirements, a *Requirement* has a *Modality*, i.e., is it an *Enhancement*, an *Obligation*, or an *Exclusion*. An enhancement indicates that the requirement may be implemented by the system, an obligation that the system has to implement the requirement, and an exclusion that the system must not implement the requirement. Next, a *Requirement* has a *Scope*. The *Scope* can be either the *System*, a *Function*, or a *Component*.

An *Availability Requirement* is a *Requirement*. It has a *Constraint*, one or more *Metric Definition(s)*, and one or more *Failure Definition(s)*. A constraint contains the actual quantification of the availability requirement, as for example "> 99.9999%".

A metric definition defines a specific metric and refers to a *Failure Mode*. We furthermore included a list of predefined metric definitions according to Junker [2016]:

- *Point Availability* is the probability of a system to be operational at a specific point in time,
- *Uptime* is the expected number of discrete time-units in an interval in which the system operates failure-free,
- *Downtime* is the expected number of time-slots in an interval where the system exhibits failure, and
- *Interval Availability* is the expected ratio of uptime in some observation period.
- The content elements *Time-Slice Availability*, *Time-Slice Uptime*, *Time-Slice Downtime*, and *Time-Slice Interval Availability* are analogous to *Point Availability*, *Uptime*, *Downtime*, *Interval Availability*, respectively, but based on time slices. Time slices are used to model the evaluation of these metrics for a certain time interval (see Junker [2016]).

Finally, a *Failure Definition* defines a *Failure Mode* on a set of *Channels*. A failure mode indicates a *Failure Type*. According to Junker [2016], we added four predefined failure types:

- *Omission*, i.e., the omission of a specific message,
- *Delay*, i.e., a specific message is delayed,
- *Insertion*, i.e., a specific message is inserted, and
- *Modification*, i.e., a specific message is replaced by another message.

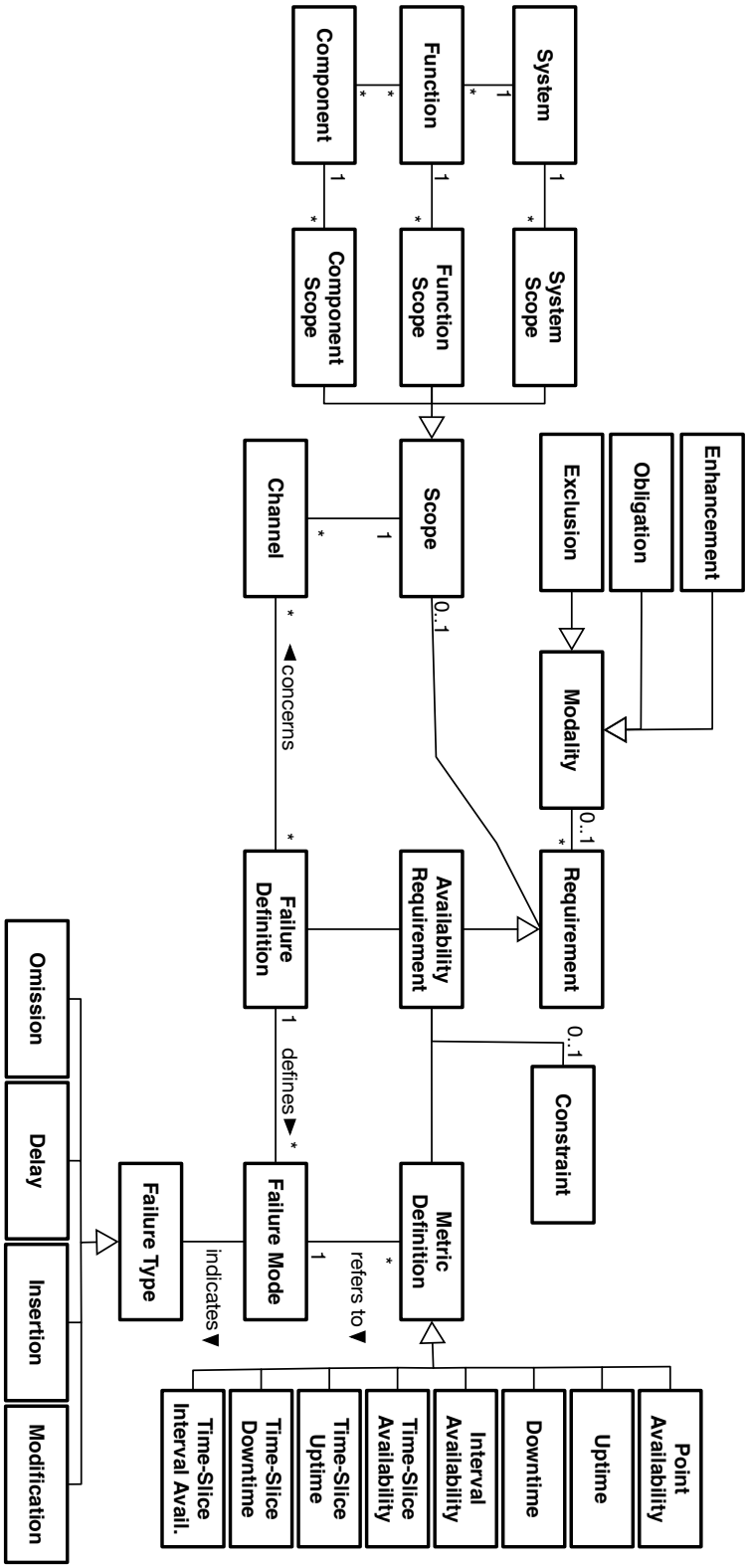


Figure 7.13.: Application to Availability Requirements: Context-independent content model of availability requirements. It shows content elements of availability requirements and their relationships.

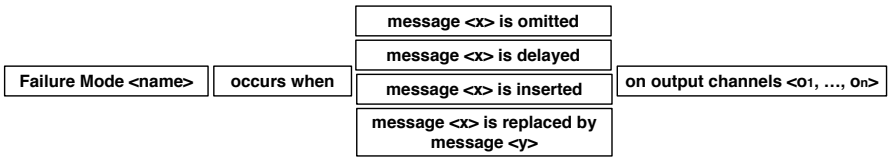


Figure 7.14.: Application to Availability Requirements: Sentence Patterns for the definition of availability failure modes.

7.5.2. Operationalization

The result of this part is an operationalization for practitioners, i.e., a means to specify availability requirements. To achieve this, we propose to derive a set of sentence patterns from the context-dependent content model (see Part ④ in Fig. 7.2). In particular, for each of the content elements in the content model, we derive a *sentence fragment*. The sentence fragment is intended to represent the meaning of the content element as closely as possible. Finally, we merge these fragments into sentences. The result of this part is a set of sentence patterns for the specification of requirements concerning the quality attribute.

For availability requirements, we iterated through the set of content items in a pair of researchers and discussed how to adequately represent this content element in terms of a sentence fragment. We derived two different types of sentence patterns:

1. Sentence patterns for the specification of availability failure modes (see Figure 7.14).

2. Sentence patterns for the specification of availability requirements (see Figure 7.15).

To create an availability requirement, we first need to define the respective failure modes. Then, we can create availability requirements. Sentences can be specified from left to right, while selecting one of the sentence fragments and replacing the variables in angle brackets. Exemplary sentences for the specification of availability modes are

> Failure Mode m_1 occurs when message m is omitted on output channel o_1.
> Failure Mode m_2 occurs when message m' is delayed on output channel o_2.

and for the specification of availability requirements

> The system must have a Point availability at time point 100 of > 99.99% with respect to failure modes m_1 and m_2.

7.6. Discussion

In this section, we discuss limitations and threats and direct implications of our approach.

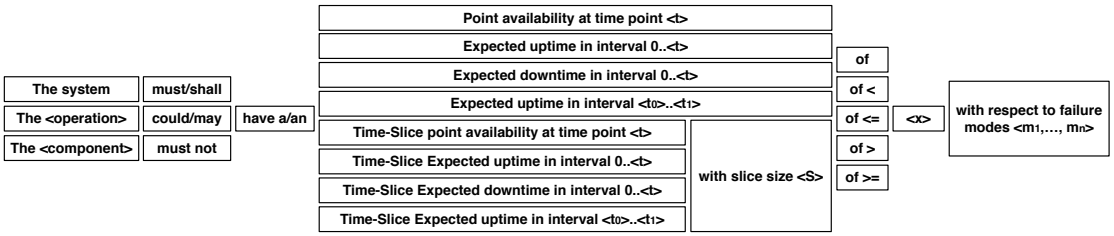


Figure 7.15.: Application to Availability Requirements: Sentence Patterns Availability Requirements.

7.6.1. Limitations and Threats

The quality of the results of our approach depends on how the individual parts are performed. Furthermore, all parts require a high amount of creative and qualitative work and thus may be error-prone. To mitigate this threat, we provided guidance in this chapter that shows how to perform the individual parts on the example of performance and availability requirements. We described how we performed the individual parts and the respective results in detail and provided hints how to ensure quality.

Context-dependent Definition

In the first part of our approach, there are some threats that affect the generalizability and applicability of the results. The initial collection of literature may miss some important work, the extraction of models may miss models or include unimportant models, and finally the coding and consolidation of the models may lead to inconsistent or inadequate models. We try to mitigate these threats by using a structured and reproducible approach (e.g., a structured literature review) and by performing the extraction and coding steps in a pair of researchers (researcher triangulation). Furthermore, we suggest to validate the resulting models with quality requirements from practice or perform validating interviews with practitioners.

Precise Definition

The goal of the second part is to create a precise definition such that we reduce misunderstandings. We propose to use either a glossary or a formal definition by means of a system modeling theory. However, in both cases, it is a challenging and creative activity and the individual content elements can be contradictory or inadequate. To mitigate this, we propose to perform a validation in form of interviews with researchers as well as with practitioners.

Context-dependent Customization

The result of this part is dependent on how the customization is performed. We propose to use activity-based quality models that try to make the relation between activities, artifacts, and quality attributes explicit. However, the quality of the results still depends

on the level of detail and adequacy of the activity-based quality model. For performance requirements, we build our customization based on the activities for testing as described in the RUP. However, the description of these activities was on a very high level of detail, and thus, we discussed each activity in a pair of researchers. In summary, to mitigate this threat, we propose to either use a detailed activity-based quality model or perform a cross validation or researcher triangulation.

Operationalization

The creation of sentence patterns is straight-forward. However, the quality of the overall approach depends on how well practitioners can apply the sentence patterns to requirements and are how much they are willing to use the patterns. To mitigate this, we propose to validate the resulting patterns with quality requirements in practice and furthermore conduct interviews with practitioners concerning their willingness to use the patterns.

7.6.2. Analyses of the Content of Quality Requirements

Besides the assessment of completeness, one can further leverage our approach to analyze the content of quality requirements in practice. Our approach results in a context-independent content model for a given quality attribute and in a context-dependent content model for that attribute. The context-independent content model provides a general definition of the content elements of the quality attribute and the context-dependent model provides a justification for each content model.

We can now analyze textual quality requirements and map the content elements found in the requirements to the content model. If we have a sufficiently large data set, we can now analyze observations and draw conclusions about the content elements of quality requirements in general. For example, a common point of view of quality requirements is that they are cross-functional and consider the system as a whole. When analyzing performance and also availability requirements, we also included the scope of a requirement in the content model. This allows us to quantitatively analyze the distribution of the scope of performance requirements found in practice.

7.6.3. Implications for Industry

Our approach is a step towards increasing the completeness of quality requirements. Not only the operationalization via sentence patterns could be easily implemented in a requirements authoring or management tool. Such a tool may provide instant feedback to the requirements engineer about missing or optional content elements, similar to requirements smells [Femmer et al., 2014a,b, 2016; Vogelsang et al., 2016]. Furthermore, the tool might check the terms used in a requirement with respect to an underlying domain model. The tool could then uncover terms that are neither part of the consolidated terminology nor defined through the pattern semantics.

An additional benefit of our approach is that it makes content in natural language requirements explicit and traceable through content elements. This allows connecting

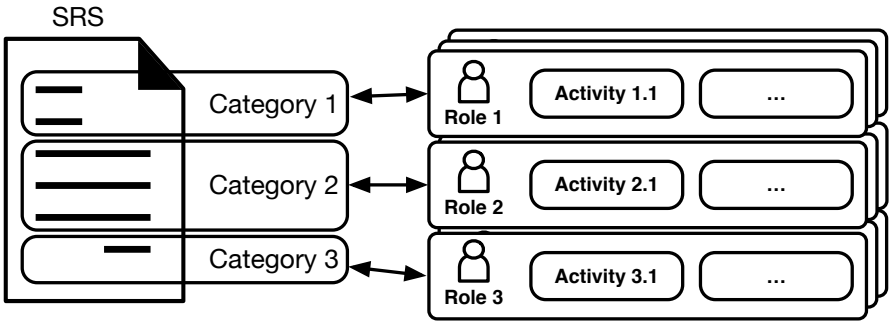


Figure 7.16.: Ideal relationship between categories of a requirements categorization and activities and roles of a process.

specific content elements of requirements with specific content elements in related artifacts such as test cases or components within the implementation. Updates within requirements may then be propagated directly to corresponding test cases for example, this makes maintenance activities more efficient and effective.

7.6.4. Usage of Requirements Categorizations in the Development Process

In this section, we discuss the relationship between categories of a categorization and activities and roles of a development process. A categorization (in general) shall clearly and unambiguously categorize elements in categories according to clearly defined arguments. Furthermore, and even more important, a categorization should have a clearly defined purpose. For requirements categorizations, this means—from a practical perspective—a categorization should categorize requirements in a way such that the activities that are performed with the requirements can be aligned according to the categories. Thus, ideally, for each of the categories, there exists a corresponding set of activities and roles for that category. As a consequence, individuals that perform an activity in a specific role can focus on the requirements from those categories that are related to the activity. This means, for example, that the performance analyst can focus on performance requirements. Figure 7.16 shows this relationship: on the left-hand side, a (software) requirements specification (SRS) is categorized in three categories and on the right-hand side, exemplary roles and activities are shown.

To better understand the relationship between categories, activities, and roles, we discuss the relationship of (exemplary) *functional requirements* and (exemplary) *performance requirements* with respect to the so called *Technical Processes* of the ISO/IEC 15288-2008 [2008]. Figure 7.17 shows these processes. We discuss this relationship along the ISO/IEC 15288-2008 [2008], as it establishes a common process framework for describing the life cycle of (software) systems. It defines a set of processes and associated terminology for the full life cycle of the system.

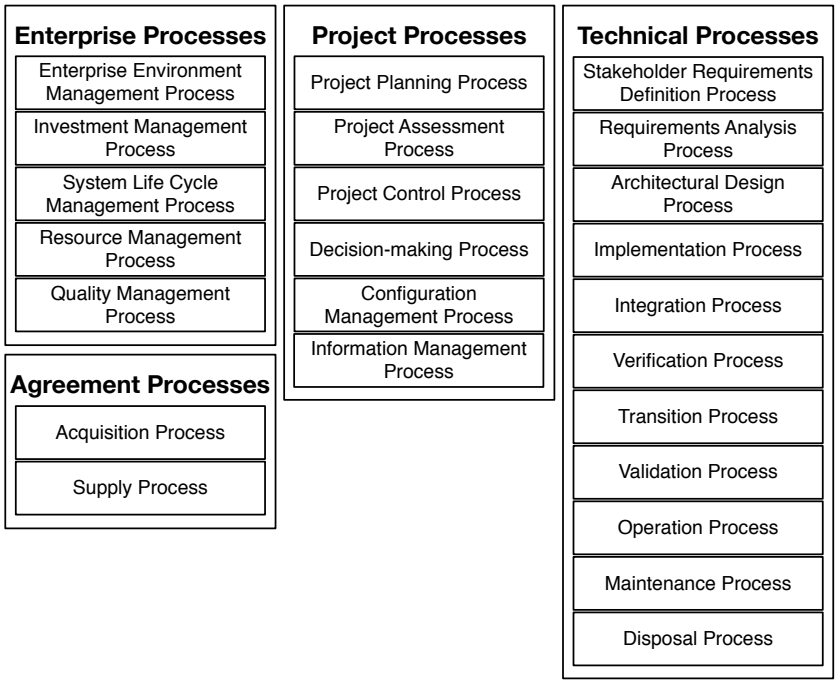


Figure 7.17.: Process groups and processes of the ISO/IEC 15288-2008 [2008].

In our investigation of how practitioners handle requirements in Chapter 4, we found that in the two phases *testing* and *architecture/design* the development activities for QRs differ most from the activities for FRs. Thus, we focus in the following discussion on the corresponding two processes, i.e., the *Architectural Design Process* and the *Verification Process*.

Architectural Design Process The purpose of the *Architectural Design Process* is to synthesize a solution that satisfies system requirements. It consists of the following three activities:

1. Define the architecture In this activity, appropriate logical architectural designs are defined, system functions are partitioned and allocated to elements of the system architecture, derived requirements for the allocations are generated, and interfaces between system elements and interfaces at the system boundary with external systems are defined and documented [ISO/IEC 15288-2008, 2008].

2. Analyze and evaluate the architecture In this activity, the resulting architectural design is analyzed to establish design criteria for each element. Moreover, it is determined, which system requirements are allocated to operators and whether hardware and software elements that satisfy the design and interface criteria are available

off-the-shelf. Finally, alternative design solutions are evaluated [ISO/IEC 15288-2008, 2008].

3. Document and maintain the architecture In this activity, the selected physical design solution is specified as an architectural design baseline in terms of its functions, performance, behavior, interfaces and unavoidable implementation constraints, the architectural design information are recorded, and mutual traceability between specified design and system requirements is maintained [ISO/IEC 15288-2008, 2008].

Discussion In these activities, there is a substantial difference, if we consider a functional requirement, as for example the requirement *"the user shall be able to delete an item from the shopping basket"*, or a performance requirement, as for example the requirement *"the system must have an average processing time of < 10 ms between event receiving a delete request and event answering a delete request, when under an average load"*.

For the functional requirement, the logical architectural design could be extended with a component at the system border which takes care of managing, and, in particular, deleting users. Analyzing and evaluating this requirement is straight forward. However, in case of the performance requirement, it is not obvious how to define an appropriate architecture that fulfills the requirement, as it may influence the architecture as a whole. Thus, we could define several candidate architectures and then perform a *performance analysis* (see for example [Balsamo et al., 2004; Bianchi, 2000; Broy et al., 2011; Jain, 1990; Wandeler et al., 2006]). A performance analysis is usually conducted in the early stages of the architecture/design and analyzes whether or not a particular distribution of functionality over a proposed system decomposition (the so-called system architecture) will meet the performance requirements. This is a difficult problem because, in this stage of the software life cycle, there are still many unknowns that might have great impact on the system performance [Wandeler et al., 2006]. Still, performance analysis is required as early as possible in the software life cycle [Wandeler et al., 2006], as performance problems often result from early design choices [Balsamo et al., 2004]. Furthermore, since performance is a runtime attribute of a software system, performance analysis requires suitable descriptions of the software runtime behavior. For example, finite state automata and message sequence charts are largely used description techniques [Balsamo et al., 2004].

Verification Process The purpose of the *Verification Process* is to confirm that the specified design requirements are fulfilled by the system. It consists of the following two activities:

1. Plan verification In this activity, the strategy for verifying the system entities throughout the life cycle is defined, the verification plan based on system requirements is defined, and potential constraints on design decisions are identified and communicated [ISO/IEC 15288-2008, 2008].

2. Perform verification In this activity, it is ensured that the enabling system for verification is available and associated facilities, equipment and operators are prepared to conduct the verification, the verification is conducted to demonstrate compliance to the specified design requirements, the verification data on the system is made available, and the verification, discrepancies and corrective action information is analyzed, recorded and reported [ISO/IEC 15288-2008, 2008].

Discussion Again, in these activities, there is a substantial difference, if we consider a functional requirement or a performance requirement. For a functional requirement like *"the user shall be able to delete an item from the shopping basket"*, the verification plan and the actual verification is straight forward, i.e., the system needs to be set up, items need to be added to the basket and then an item needs to be deleted. However, with a performance requirement like *"the system must have an average processing time of < 10 ms between event receiving a delete request and event answering a delete request, when under an average load"* both activities are more complicated. In this case, we need to set up a suitable hardware against which to test the processing time, as the actual processing time depends on the hardware of the system. Furthermore, as the requirement specifies an average load, we need to consider how to simulate a suitably heavy load in the test environment. For throughput requirements, this is even more complicated: We need an automated way to generate a high volume of requests, as manually generating these would be logistically impossible (in most cases). For performing the measurement, we need to set up a measurement that accurately measures the processing time.

Conclusion & Outlook In this section, we argued that a requirements categorization should categorize requirements in a way such that the activities that are performed with the requirements can be aligned according to the categories. We exemplarily discussed the difference in the activities of the *Architectural Design Process* and the *Verification Process* between functional and performance requirements. We argued that there are substantial differences between functional and performance requirements in performing these activities. Traditional software development methods focus on functional correctness and introduce performance requirements later in the process. However, as performance problems may have a severe impact on the overall project success, there has been a growing interest and several approaches to, e.g., early software performance predictive analysis, have been proposed. Balsamo et al. [2004] present a comprehensive review of recent research in the field of model-based performance prediction at software development time. They argue from their results, that the software performance predictive process is based on the availability of software artifacts, as for example requirements, software architectures, and specification, that describe suitable abstraction of the final software system. Moreover, they argue that performance analysis requires suitable descriptions of the software runtime behavior as performance is a runtime attribute of a software system. For the evaluation of performance models, analytical methods and simulation techniques can be used to get performance indices, as for example throughput, utilization, response time.

Thus, we argue that we can drastically reduce the amount of requirements that need to be processed, i.e., read, understand, and analyzed, for performing an activity, if we make the relation between requirements categories, activities, and roles explicit. An explicit relation allows individuals performing an activity in a specific role to focus on the requirements from the respective category. Thus, the sheer amount of requirements that need to be processed for performing the activity is reduced to the requirements of the corresponding category.

However, up until now, the concrete relationships between specific requirements categorizations and the development activities and their roles is still an open question. To derive concrete relationships, we propose an approach with the following steps:

Detailed Analysis of Development Activities As a first step, we propose to analyze the activities of common process frameworks, as for example the ISO/IEC 15288-2008 [2008] or the RUP [Jacobson et al., 1999]. The goal of this step is to create an explicit mapping of requirements categories and activities that are performed with them. To reach this goal, we propose to analyze the descriptions of the individual steps of the activities and then to discuss whether the steps are performed in different ways for different categories. If there are different tasks performed with different categories, we propose to make this difference explicit. The result of this step is a refined model of the activities, where each activity is explicitly mapped to the categories of a categorization. However, we found that the descriptions of both, the ISO/IEC 15288-2008 [2008] as well as the RUP, are rather high level, and thus the results of this step bear the threat of being subjective and incomplete.

Interview Studies with Participants from Industry To mitigate the subjectivity and incompleteness threat of the first step and to further detail the relationship between development activities and requirements categories, we propose to further conduct a series of interview studies. In each study, we propose to select a participant that takes a role (for a given activity) in their daily work. Then, we propose to present requirements of different categories to the participant and ask how the participant usually performs the tasks for the requirement. The result of this step is an activity model where the relationship between activities and categories is further refined and backed by evidence from practice.

The result of this approach is an activity model in which the relationship between activities and categories is made explicit. In order to be useful for the process, we then require that

1. all categories are mapped to at least one activity, and
2. activities are not linked to a high number or all categories.

In the first case, i.e., the resulting model contains categories that are not linked to any activity, which means that requirements of that category are not used in any development activity, we may argue that we should think about changing the categorization or about changing the process. The same holds for the second case, i.e., there are activities that

are linked to all categories. In this case, either the activity in deed needs requirements from all categories or the categorization may be not useful for selecting requirements for that activity. Again, in this case, we should think about changing the categorization or about changing the process.

However, we only discussed the relationship and presented a possible approach to make these relationship concrete. We leave it as subject to future research (see Chapter 10) to instantiate and evaluate our approach. A long-term vision that follows from this is that we might be able to better integrate requirements categorizations into a holistic software and system development process in the future. Such an integration would yield, for instance, seamless modeling of all properties associated with a system. The benefits of such an integration include that specific categories would not be neglected during development activities, as it is too often current state of practice; from an improvement in the traceability [Eder et al., 2014] of requirements over an improvement of possibilities for progress control to an improvement of validation and verification.

7.7. Related Work

There is a variety of work on requirement patterns in RE. Franch et al. [2010] present a metamodel for software requirement patterns. Their approach focuses on requirement patterns as a means for reuse in different application domains and is based on the original idea of patterns by Alexander [1979], i.e., each pattern describes the core of a solution of a problem that occurs over and over again. In particular, the PABRE framework contains a catalogue of 29 QR patterns [Renault et al., 2009], 37 non-technical patterns [Palomares et al., 2012], and a method for guiding the use of the catalogue in RE [Franch et al., 2013]. Their approach for creating the patterns catalogue is similar to ours, as it is also based on requirements literature and a content analysis. However, they provide solutions for recurring problems while our sentence patterns provide a means for the specification of customized requirements.

Supakkul et al. [2010] present four kinds of NFR patterns for capturing and reusing knowledge of NFRs and apply these patterns in a case study. Their patterns and, in particular, the *objective pattern* can be used to identify important NFRs for a context or capture a specific definition of an NFR from the viewpoint of a stakeholder. Thus, their patterns define important content elements of a quality attribute in terms of soft goals, which is similar to our context-dependent content model [Chung et al., 2012; Mylopoulos et al., 1992]. Our approach provides a structured way to define and customize these content elements and also provides sentence patterns to specify requirements. However, their patterns can be used to define the specific quality attribute but furthermore provide solutions and alternatives and thus go one step further into the architecture or design of a system. Our approach focuses on definition, customization, and concretization of requirements concerning a specific quality attribute.

Withall [2007] presents a comprehensive pattern catalogue for natural language requirements in his book. The pattern catalogue contains a large number of patterns for different types of requirements. In contrast to their work, in our approach, we derive patterns from

literature and customize them to a specific application context. De Almeida Ferreira and Rodrigues da Silva [2013] introduce RSL-PL, a language for the definition of requirements sentence patterns. Their pattern definition language can be used to represent our sentence patterns.

Kopczyńska and Nawrocki [2014] present a method for eliciting non-functional requirements, which is composed of a series of brainstorming sessions driven by the ISO/IEC 25010-2011 [2011] quality sub-characteristics. Elicitation is supported by Non-functional Requirements Templates (NoRTs), which are statements that require some completion to become a well-formulated NFR. Similar to our sentence patterns, the authors differentiate between core parts, parameters, and optional parts within the templates. The sentence patterns derived by our approach are additionally adapted to specific classes of quality requirements.

Mylopoulos et al. [1992] propose a comprehensive framework for representing and using QRs in the development process. Similar to our approach, they propose a means to integrate QRs in the development process, however, they do not provide a structured approach for explicitly stating the content elements for specifying requirements concerning quality attributes and do not provide a means for specifying QRs.

7.8. Conclusions

The goal of this chapter was to reach our third objective. In particular, our goal was to develop an approach for defining, specifying, and integrating quality requirements based on a system model and to assess whether it is applicable in practice. To this end, we provided an approach that—given a quality attribute as input—provides a means to precisely and explicitly define the content elements that are needed to specify requirements concerning this quality attribute, and provides a means for practitioners to specify these requirements for a given organizational context based on sentence patterns. The approach consists of four parts:

1. **Context-independent Definition:** Relevant content elements are identified by means of qualitative literature analysis and coding.
2. **Precise Definition:** The resulting content elements are precisely defined by e.g., a glossary or formalization by means of a mapping to a system model.
3. **Context-dependent Customization:** The content elements are customized to a given organizational context by using the idea of activity-based quality models.
4. **Operationalization** Sentence patterns are used as a means for practitioners to specify requirements concerning the quality attribute.

As our main goal was to provide guidance for the application of our approach, we furthermore discussed threats to validity and lessons learnt while instantiating it for performance and availability requirements. Finally, we argue that our approach is applicable for performance and availability requirements and besides its constructive

nature, provides a means for various static analyses, as for example completeness analyses. This contribution supports (the second part of) our hypothesis, i.e., a categorization based on a system model is operationalizable for subsequent development activities.

In the next chapter, we perform an empirical evaluation of our approach with respect to its applicability and ability to uncover incompleteness.

"Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt."
— Ludwig Wittgenstein

8 Chapter

Validation of the Approach on the Example of Performance Requirements

Parts of this chapter have been previously published in the following publications:

- Eckhardt, J., Vogelsang, A., Femmer, H., and Mager, P. (2016b). Challenging Incompleteness of Performance Requirements by Sentence Patterns. In *Proceedings of the 24th International Requirements Engineering Conference (RE)*, pages 46–55 (full paper, research track, 10 pages)

Performance requirements play an important role in software development. They describe system behavior that directly impacts the user experience. Specifying performance requirements in a way that all necessary content is contained, i.e., the completeness of the individual requirements, is challenging, yet project critical. Furthermore, it is still an open question, what content is necessary to make a performance requirement complete. In this chapter, we instantiate our approach for performance requirements and conduct an empirical evaluation with respect to its applicability and ability to uncover incompleteness.

The results of the application of our approach to the quality attribute performance are already described in Chapter 7: The context-independent content model is shown

in Figure 7.8, the context-dependent content model in Figure 7.10, and finally, the operationalization by means of sentence patterns in Figure 7.11. In this chapter, we derive a notion of completeness based on the context-dependent content model and evaluate both the applicability of the approach as well as its ability uncover incompleteness with performance requirements taken from 11 industrial specifications.

In our study, we were able to specify 86% of the examined performance requirements by means of the sentence patterns for performance requirements. Furthermore, we show that 68% of the specified performance requirements are incomplete with respect to our notion of completeness. We argue that our approach provides an actionable definition of completeness for performance requirements that can be used to pinpoint to requirements that are hard to comprehend, implement, and test.

8.1. Context: Completeness of Performance Requirements

One of the most important problems in requirements engineering (RE) is incompleteness. In a survey with 58 requirements engineers from industry, Méndez and Wagner revealed that incomplete requirements are not only named as the most frequent problem in RE but also the most frequent cause for project failure [Méndez Fernández and Wagner, 2013]. Incompleteness can be considered on two levels: incomplete requirements specifications as a whole or incomplete requirements, i.e., lack of details for single requirements. In the following, we focus on the latter problem. The problem of incompleteness concerns both functional and quality requirements, such as performance requirements[15]. But what makes a performance requirement complete? Although classifications and definitions exist, it still remains unclear which content a performance requirement should contain.

To address this lack, we applied our approach to performance requirements, resulting in a context-independent and context-dependent content model for performance requirements, a clear and precise definition of the individual content elements, and an operationalization through sentence patterns. Furthermore, we derive a notion of completeness of performance requirements based on the context-dependent content model.

To evaluate our approach, we applied the sentence patterns to 58 performance requirements taken from 11 industrial specifications and analyzed (i) the applicability and (ii) the ability to uncover incompleteness. We were able to rephrase 86% of the performance requirements by means of the sentence patterns. Moreover, we found that our approach can be used to detect incompleteness in performance requirements, revealing that 68% of the analyzed performance requirements were incomplete.

In summary, we contribute: (i) a notion of completeness for performance requirements and (ii) an empirical evaluation of our approach with respect to its applicability and ability to detect incompleteness in requirements.

The remainder of this chapter is structured as follows: In Sect. 8.2, we present our notion of completeness for performance requirements. Then, we present the study design

[15] In the remainder of this chapter, quality requirements (QRs), we refer to product-related QRs, i.e., requirements that address quality characteristics of the product or system and exclude process requirements or constraints.

and results of our evaluation in Sect. 8.3 and discuss the implications in Sect. 8.4. Finally, in Sect. 8.5, we report on related work before we conclude our work and discuss future research in Sect. 8.6.

8.2. Notion of Completeness for Performance Requirements

Following the idea of an activity-based definition of quality attributes (see [Deissenboeck et al., 2007; Femmer et al., 2015]), we created the context-dependent content model in Section 7.4.3 based on development activities that stakeholders conduct with performance requirements. We identified necessary content elements that a performance requirement must contain to complete these development activities efficiently and effectively. For example, the scope of a requirement is necessary for the activity *defining a performance test*. In Figure 7.10, we marked the crucial content elements with a white background and mandatory content elements with a gray background. This results in 15 mandatory content items.

Given a performance requirement, we define the completeness of the requirements with respect to the presence of all mandatory content elements, i.e., we call a requirement complete if all mandatory content elements are present in the textual representation of the requirement. There are three cases for the presence of mandatory content elements in the textual representation of a requirement:

Case 1: The requirement does not contain the content element. For example, in case of a requirement stating *"The delay between [event A] and [event B] shall be short"*, the content element regarding the quantifier is not contained.

Case 2: The requirement **implicitly** contains the content element. With implicit, we mean that the content element is contained in the requirement, but we need to interpret the requirement to derive the content element. For example, in case of a requirement stating *"The delay between [event A] and [event B] shall typically be 10ms"*. In this case, regarding the quantifier, we can interpret "typically" as "median".

Case 3: The requirement **explicitly** contains the content element. With explicit, we mean that the content element is contained without interpretation. For example, in case of a requirement stating *"The delay between [event A] and [event B] shall have a median value of 10ms"*. In this case, regarding the quantifier, the content element is explicitly contained.

We derive the following definitions for strong and weak completeness and for incompleteness of performance requirements:

Definition (Strong Completeness of Performance Requirements)**.** *A performance requirement is* ***strongly complete****, if all mandatory content elements (w.r.t the content model) are* ***explicitly*** *contained in its textual representation.*

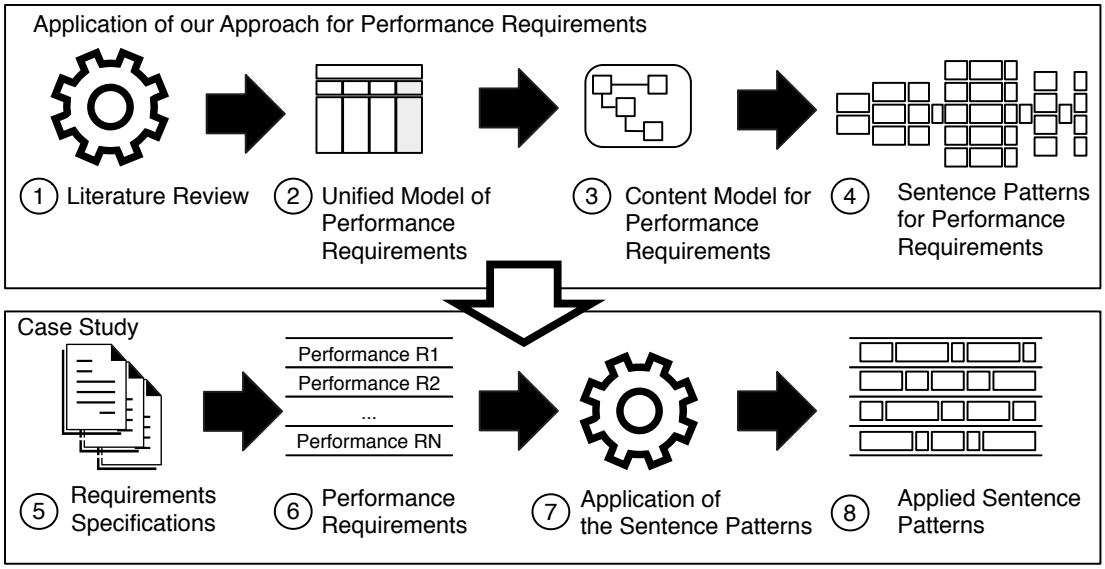


Figure 8.1.: Overview of our Research Methodology.

Definition (Weak Completeness of Performance Requirements)**.** *A performance requirement is* ***weakly complete****, if all mandatory content elements (w.r.t. the content model) are* ***explicitly or implicitly*** *contained in its textual representation.*

Definition (Incompleteness of Performance Requirements)**.** *A performance requirement is* ***incomplete****, if at least one mandatory content elements (w.r.t. the content model) is* ***missing*** *in its textual representation.*

This definition of completeness for performance requirements can be used to detect incompleteness and thus to pinpoint to requirements that are hard to comprehend, implement, and test. For example, requirements of class *incomplete* are not testable at all, requirements in class *weakly complete* need to be interpreted by the developer and/or tester and therefore bear the risk of misinterpretations, and requirements in class *strongly complete* contain all content necessary to be implemented and tested.

8.3. Case Study

In order to evaluate our sentence patterns for performance requirements, we conducted a case study with industrial performance requirements. Figure 8.1 shows an overview of our research methodology: First, we applied our approach to performance requirements (Step ①–④) and then, we collected performance requirements and applied the sentence patterns (Step ⑤–⑧). In the following, we first describe the design of the study and then report on the results.

8.3.1. Study Design

The goal of our study is to understand the applicability and ability to detect incompleteness of our sentence patterns in the context of natural language performance requirements from industrial specifications.

Research Questions

To reach our goal, we formulate the following research questions (RQs). In RQ1 and RQ2, we analyze how well our sentence patterns match performance requirements in industry.

RQ1: *To what degree can industrial performance requirements be specified by means of our sentence patterns?*

RQ2: *Can our sentence patterns be used to detect incompleteness in industrial performance requirements?*

In RQ3 and RQ4, we analyze how well performance requirements in industry match with the context-dependent content model.

RQ3: *What type of performance requirements are used in practice?*

RQ4: *What content is used in performance requirements in practice?*

Study Object

In a previous study [Eckhardt et al., 2016c], we analyzed 530 requirements that were labeled as "non-functional", "quality requirement", or a specific quality attribute in 11 industrial specifications from 5 different companies for different application domains and of different sizes. In particular, we classified each requirement according to its ISO/IEC 9126-2001 [2001] quality characteristic (e.g., *Efficiency–Time Behaviour*).

The study objects used to answer the research questions constitute of these 11 industrial specifications. We collected all those requirements that are classified as *Efficiency–Time Behaviour* or *Efficiency–Resource Utilization*. This results in 58 performance requirements in total. We cannot give detailed information about the individual performance requirements or the projects. Yet, in Table 8.1, we show exemplary (anonymized) performance requirements as far as possible within the limits of existing non-disclosure agreements.

Data Collection

To answer our research questions, we applied the sentence patterns to each performance requirement of our study object. If we were not able to apply the patterns due to missing or too vague information, we marked the requirement accordingly (e.g., the requirement "[...] No significant decrease in performance is permitted").

When applying a specific sentence fragment of a pattern, for example, the quantifier `(a | a mean | a median | a maximal | a minimal)`, there are three cases:

Table 8.1.: Exemplary performance requirements

Spec.	Requirement	Domain
S2	*The delay between [event 1] and [event 2] shall be less than 1s.*	ES (Railway)
S3	*The [system] must ensure the following average response times for specific use cases under target load:* $UC_1 < 1min$ $UC_2 < 2\ min$ ... *Note: The timing has to be considered a net time with respect to all the back-office interfaces.*	BIS (Automotive)
S6	*The delay between receiving of [message] and the update of [signal]* *Start Event: [event]* *Stop Event: [event]* *Value < 1.5 sec* *Notes It is assumed that the [signal] is required by the message received. The value indicated in this case includes additional delay for the display of the information.*	ES (Railway)

- The requirement explicitly contains the content element. In this case, we mark the resulting value as explicit. For example, in case of a requirement stating *"The delay between [event A] and [event B] shall have a median value of 10ms"*, we set the quantifier to explicit `Median`.
- The requirement implicitly contains the content element. In this case, we mark the resulting value as implicit. For example, in case of a requirement stating *"The delay between [event A] and [event B] shall typically be 10ms"*, we set the quantifier to implicit `Median`.
- The requirement does not contain the content element. In this case, we mark this sentence fragment as missing. For example, in case of a requirement stating *"The delay between [event A] and [event B] shall be short"*, we set the quantifier to missing.

The procedure was performed by the first two researchers in pair. Both have over three years of experience in requirements engineering research and model-based development research. Table 8.2 shows examples of the resulting requirements; Explicit content is marked by subscript "e", implicit content is marked by subscript "i" and missing content by subscript "m".

Data Analysis Procedures

To answer RQ1, we analyzed whether the sentence patterns can be applied for the given performance requirements. To answer RQ2, we analyzed to what degree the requirements

Table 8.2.: Data Collection: Exemplary application of the sentence patterns. Explicit content is marked by subscript "e", implicit content is marked by subscript "i" and missing content by subscript "m".

#	Original Requirement	Applied Pattern	Type	Completeness
R1	*The train door release command delivered by [component B] to [component C] must not be delayed more than 500ms by the [component D].*	The component "D"$_e$ must not$_e$ have a$_e$ latency$_e$ of >$_e$ 500$_e$ ms$_e$ between event "train door release command delivered by component B" and event "train door release command received by component C"$_i$	Time Beh.	Strongly Compl.
R2	*The delay between door close detection and authorization to depart shall be less than 500ms.*	The system$_i$ must$_e$ have$_e$ a$_e$ latency$_e$ of <$_e$ 500$_e$ ms$_e$ between event "door close detection " and event "authorization to depart"$_i$	Time Beh.	Weakly Compl.
R3	*Cycle of position reports: Value > 5s. Notes: This performance defines the maximum rate for sending of position reports.*	The system$_m$ must$_m$ have a$_e$ transaction rate$_i$ of <$_i$ 1/5$_i$ position report$_e$ per s$_e$	Throughput	Incomplete
R4	*No significant decrease in performance is permitted [...]*	N/A	N/A	Incomplete

are *strongly complete*, *weakly complete*, or *incomplete* with respect to our notion of completeness. To answer RQ3, we analyzed the distribution of the requirements with respect to their performance requirement type. To answer RQ4, we analyzed the mapping between the original requirements and the content elements in our context-dependent content model. We perform this analysis for content elements that are applicable for all performance requirements (like *Scope*) and also for each of the individual performance requirement types.

8.3.2. Study Results

RQ1: Applicability of our Sentence Patterns

In total, we could apply the patterns to 50 of the 58 performance requirements. We could not apply the patterns to 8 requirements, because of missing or too vague information (see for example, requirement R4 in Table 8.2). Thus, in total, 86% of the requirements can be expressed by means of our sentence patterns for performance requirements.

The remaining 14% of the requirements are either internal performance requirements (e.g. *"System shall support and leverage 64-bit Hardware and Operating Systems for scaling up"*), non-specific requirements, or just high-level goal descriptions. We explicitly excluded internal performance requirements from our content model and could not apply our sentence patterns to the others, as they leave to much room for interpretation.

> **Quantitative results of RQ1:**
> 86% of the performance requirements can be expressed by means of our sentence patterns.

RQ2: Benefits of our Approach

In total, 18% of the 50 requirements are *strongly complete*, 32% are *weakly complete* and 68% are *incomplete*.

Analyzing the distribution in more detail, the application of the sentence patterns for the 50 sentences resulted in 396 sentence fragments. Figure 8.2, shows a partially aggregated view on the results for the mandatory content elements: *Value* aggregates *Time Value*, *Change Value*, and *Capacity Value*. *Property* aggregates *Time Property*, *Throughput Property*, and *Capacity Property*. As shown in the figure, most requirements (93%) specify the value explicitly. This is as one would expect for performance requirements, as the value specifies the specific time or resource bound for the requirement. In contrast to this, the scope of only 48% of the requirements is explicitly stated in the requirement, but can be interpreted for 48% and is missing for 4% of the requirements. This might be no problem for most requirements, but not explicitly stating the scope leaves room for interpretation and bears the risk of misunderstanding for which functions of a system a performance requirement holds.

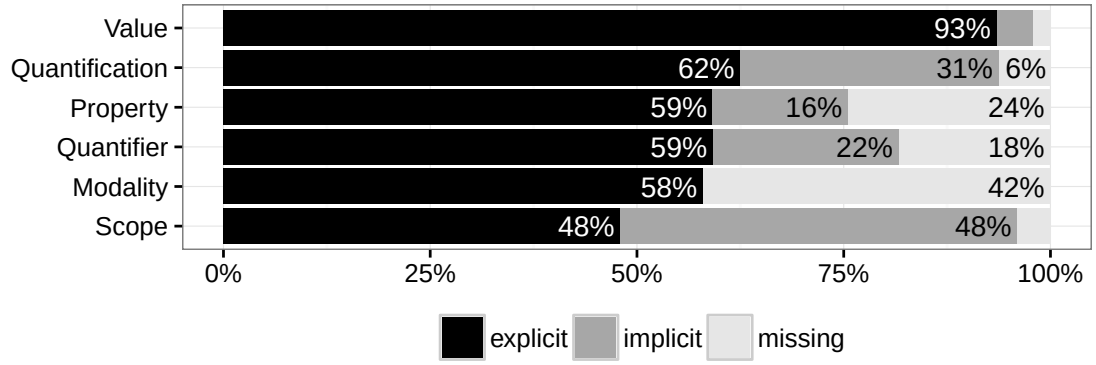


Figure 8.2.: RQ2: Completeness of the original requirements w.r.t. the mandatory content elements. *Value* aggregates *Time Value*, *Change Value*, and *Capacity Value*. *Property* aggregates *Time Property*, *Throughput Property*, and *Capacity Property*.

Quantitative results of RQ2:
18% of the 50 requirements are *strongly complete* and 32% are *weakly complete*. The remaining 68% are incomplete with respect to our notion of completeness.

RQ3: Performance Requirement Type

In total, 35 out of the 50 performance requirements are of type *Time Behavior* (70%), 13 of type *Capacity* (26%) and 2 of type *Throughput* (4%).

Quantitative results of RQ3:
70% of the performance requirements concern *Time Behavior*, 26% *Capacity*, and only 4% *Throughput*.

RQ4: Performance Content

Figure 8.3 shows the results of RQ4. In particular, Figure 8.3a shows the distribution among the concepts of all requirements, Figure 8.3b shows the distribution among time behavior requirements, and Figure 8.3c shows the distribution among capacity requirements. Note that we do not detail the results for throughput requirements, since only 4% of the requirements were of this type.

In contrast to the prevailing opinion that QRs are cross-functional, the scope of only 58% is the whole system, for 34% it is a function and for 8% a component. For time

behavior requirements, the percentage of requirements having a function as scope (49%) even rules out the percentage of requirements having the system as scope (46%). In contrast to this, for capacity requirements, 85% of the requirements specify the system as scope and only 15% a component as scope. Therefore, one could argue that while capacity performance requirements are mostly cross-functional, this is not necessary the case for behavioral performance requirements.

Furthermore, it stands out that most requirements (98%) are an obligation and only 2% an exclusion.

8.4. Discussion

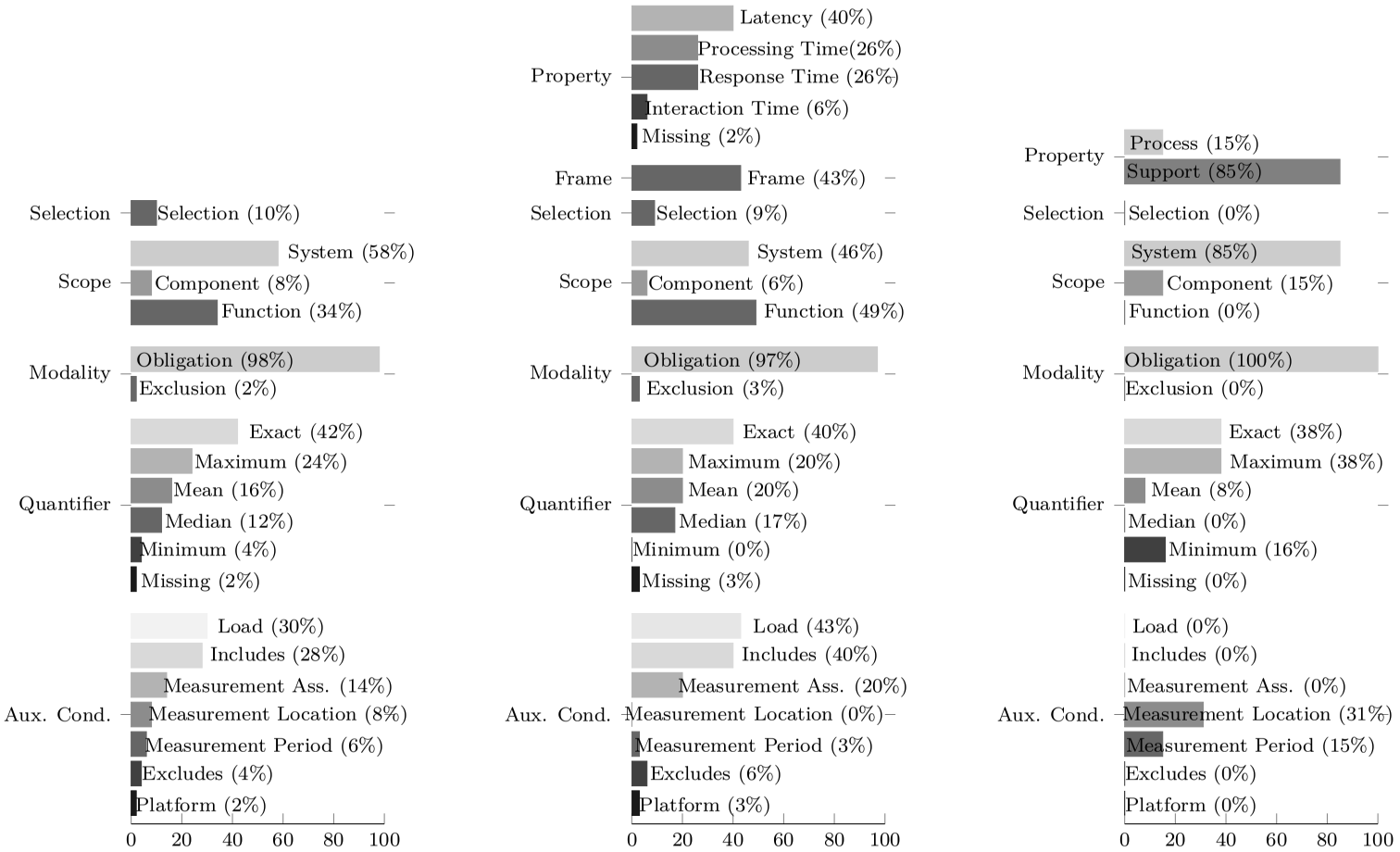
From the presented results, we conclude that our proposed sentence patterns for performance requirements are applicable to performance requirements documented in practice. Furthermore, we argue that our approach provides a helpful and actionable definition of completeness for performance requirements that can be used to detect incompleteness and thus to pinpoint to requirements that are hard to comprehend, implement, and test.

We draw these conclusions by connecting the major results of our evaluating case study: We were able to apply our sentence patterns to 86% of the requirements in a large set of natural language performance requirements from practice. Our definition of completeness is derived from 15 mandatory content elements. Neglecting or implicitly stating one of these content elements has a negative impact on subsequent development activities (e.g., implementation or testing). With respect to our notion of completeness, from the investigated requirements, only 18% were complete (*strongly complete*), 32% contained mandatory content elements only implicitly (*weakly complete*), and 68% neglected at least one mandatory content element (*incomplete*). We argue that requirements of class *incomplete* are not testable at all, requirements in class *weakly complete* need to be interpreted by the developer and/or tester and therefore bear the risk of misinterpretations, and requirements in class *strongly complete* contain all content necessary to be implemented and tested.

Besides the assessment of completeness, we made some unexpected observations that question some common views onto performance requirements and QRs in general. A common point of view for QRs is, for example, that QRs are cross-functional and consider the system as a whole. We were surprised to see that in our study the scope of 42% of the requirements that we examined was “component” or “function” (see Figure 8.3). This means that, at least in the analyzed specifications, several requirements are actually framed by functions or specific components and not always with respect to the whole system. Especially for time behavior requirements, a majority of the requirements were associated with a function. However, for testing or verification, it might still be necessary to consider the system as a whole.

8.4.1. Implications for Academia

We consider the (re)definition of individual quality attributes based on their impact to development activities as beneficial for operationalizations. Activity-based quality models



(a) Concepts of all requirements.

(b) Concepts of time behavior requirements.

(c) Concepts of capacity requirements.

Figure 8.3.: RQ2: Distribution among the performance concepts. Concepts of throughput requirements are excluded as we only analyzed two.

(e.g., [Deissenboeck et al., 2007; Femmer et al., 2015]) provide frameworks to define and operationalize quality attributes such as completeness. In our approach, we derived a context-dependent content model for performance requirements based on the question which content is necessary to perform specific activities. This approach leads to quality assessments that can directly be related to activities. It would be interesting to apply a similar approach to assess the completeness of other quality attributes.

Our approach captures the content of a requirement as a model. Building such models for industrial requirements allows reasoning about several statements that are presumed to be common knowledge about QRs. For example, the assertion that QRs are cross-functional and affect the whole system is challenged by the fact that a reasonable share of examined requirements regarded the scope "function" or "component" instead of "system".

8.4.2. Implications for Industry

Our results suggest that natural language performance requirements in practice are, to a large extent, incomplete with respect to our notion of completeness or at least need to be interpreted to be implemented and tested. Our approach is a step towards increasing the completeness of performance requirements. The operationalization via requirement patterns could be easily implemented in a requirements authoring or management tool. Such a tool may provide instant feedback to the requirements engineer about missing or optional content elements. Furthermore, the tool might check the terms used in a requirement with respect to an underlying domain model to uncover terms, the reader must interpret because the term is not part of the consolidated terminology.

An additional benefit of our approach is that it makes content in natural language requirements explicit and traceable through content elements. This allows connecting specific content elements of requirements with specific content elements in related artifacts such as test cases or components within the implementation. Updates within requirements may then be propagated directly to corresponding test cases for example, making maintenance activities more efficient and effective.

8.4.3. Limitations of our Sentence Patterns for Performance Requirements

In total we were able to apply our sentence patterns to 86% of the 58 natural language performance requirements from practice. That means that we were not able to apply our sentence patterns to 8 requirements (14%). Table 8.3 shows the list of the requirements to which the sentence patterns were not applicable. It furthermore shows reasons why the sentence patterns were not applicable. When we take a closer look at the requirements and the reasons, we have three major reasons why we were not able to apply the sentence patterns:

1. The requirement is not a software/performance requirement (c.f., R4, R8 and partially R1 and R5 in Table 8.3). R4 restricts the maximal (physical) size of a component, R8 specifies the warranty period and replacement period, and R1 and

R5 describe the utilization of specific hardware. Our sentence patterns are intended to capture the contents that are needed to specify performance requirements and, thus, we were not able to apply the patterns.

2. The requirement is rather a goal than a concrete and testable requirement (c.f. R1 and R5 in Table 8.3). R1 as well as R5 describe that the system shall *support* and *leverage* a specific hardware for *scaling up*. These two requirements are rather a high level goal, i.e., *The system shall scale up*, and do not describe a concrete and testable requirement. Our sentence patterns are intended to be used to specify performance requirements with a focus on testing the requirement. Therefore, we were not able to apply the patterns in these cases.

3. The requirement is too vague and/or contains references (c.f., R2, R3, R6, R7 in Table 8.3). R2 specifies that no *significant decrease in performance* is permitted while R3, R6, R7 permit a *x-fold decrease in performance* in specific cases. When reading the singular requirement, the reader cannot clearly understand what *significant* means and furthermore, without any reference to the "normal" performance of the system, the reader cannot understand the meaning of an *x-fold* decrease. Moreover, the requirements contain the word *performance*. This is a very broad term and without a clear definition, the reader cannot understand what this means. In summary, in these cases, the requirements cannot be understood as a singular requirement, as they are either too vague or contain references to definitions and values. Thus, we were not able to apply our patterns as we require this information to be present.

In summary, the first and second reason why we could not apply our sentence patterns concern the type of the requirement: Either it is not a software requirement or it is a high level goal. The third reason why we could not apply them is that the requirements are too vague or contain references to definitions and values and cannot be understood as singular requirements. The latter reason is a problem of sentence patterns in general, as sentence patterns require requirements to be singular. Thus, they are not allowed to reference other information in the requirements specification. This can, however, be overcome with an explicit glossary where each term, as for example *significant decrease*, is precisely defined.

8.4.4. Threats to Validity

We assess the completeness of performance requirements by mapping natural language requirements to a content model that we derived from literature. An assessment whether a requirement is complete or incomplete is therefore always relative to the notion of completeness used. If the content model that we used for this study itself is incomplete or too strict, the results about the completeness of examined requirements in practice would be misleading. A less strict content model, that defines less mandatory content elements, would result in more requirements that are considered complete. From our point of view, a "good" definition of the content model should be derived from the activities that need

Table 8.3.: List of requirements for which we were unable to apply the sentence patterns to.

Id	Requirement	Reason
R1	[The] system shall support and leverage 64-bit Hardware and Operating Systems for scaling up.	Patterns not applicable, because the requirement is rather a goal than a concrete and testable requirement. Furthermore, the requirement talks about utilization of specific hardware.
R2	No significant decrease in performance is permitted in this case [case: applications connected via broadband]	Patterns not applicable, because too vague. E.g., what does *significant decrease* mean?
R3	Remote clients running general applications connected via an ISDN phone line greater than 64 kbps. A 3-fold decrease in the connected client performance is permitted.	Patterns not applicable, because too vague (*performance*) and *3-fold decrease* is a reference without making the referenced value explicit.
R4	The size of [component X] must not exceed the values of 620mm x 1250mm x 450mm.	Patterns not applicable, because not a software requirement.
R5	[The] system shall support and leverage multi-core CPUs for scaling up.	Patterns not applicable, because this is rather a goal than a concrete and testable requirement. Furthermore, the requirement talks about utilization of specific hardware.
R6	Baud rates (e.g. GSM) less than 10 kbps. A 8-fold decrease in performance permitted.	Patterns not applicable, because too vague and *8-fold decrease* is a reference without making the referenced value explicit.
R7	Remote clients running with basic applications for system operation and visualization connected via analog leased or PSTN lines greater than 33 kbps. A 5-fold decrease in performance is permitted.	Patterns not applicable, because too vague and *5-fold decrease* is a reference without making the referenced value explicit.
R8	Typical warranty period being, for example, of two years. Typical replacement period is 5 years.	Patterns not applicable, because not a software requirement.

to be performed based on the requirements (see [Femmer et al., 2015]). We tried to justify all mandatory content elements in our content model by considering development activities that are not or hardly possible without this content.

Furthermore, (strongly complete) sentences created by our patterns still may be ambiguous and thus subject to interpretations. This may be the case as some sentence fragments, such as the time property *processing time*, may have a different meaning depending on the context. To mitigate this threat, we suggest to assign a context specific meaning for those sentence fragments and make this meaning explicit by means of for example a glossary. The same holds for domain objects like *concurrent users*.

A major threat to the internal validity is that our results and conclusions strongly rely on the classification and translation of requirements into patterns, which was performed by the authors of this study. To mitigate biased classifications and pattern translations, we performed the classification in a pair of researchers. A third researcher afterwards reviewed the resulting patterns and challenged the reliability of the classification. This lead to two rounds of refinement of classification and patterns.

Another threat that might influence the results of our case study is that we examined only requirements that we identified as *performance* requirements in a former study [Eckhardt et al., 2016c]. With this selection procedure, some relevant performance requirements might have been missed or irrelevant ones might have been included.

We base our evaluation on a set of 58 performance requirements that we extracted from 11 industrial specifications from 5 different companies for different application domains and of different sizes with a total of 530 requirements. That means that performance requirements were only one part of the specification and accounted only for 11% of all requirements. It is possible that there exist additional documents specifically made for performance requirements, which may refine the examined requirements for specific purposes such as testing. Additionally, it might also be possible that companies have special teams or departments for implementing or testing performance requirements. It is possible that these teams just take the general performance requirements from the examined specifications as an input and translate them to requirements that are more complete w.r.t. our notion of completeness. We are not aware of such additional documents or teams in our cases.

There are few threats that affect the generalizability of our results and conclusions: We have based our context-independent content model on 12 existing classifications that we identified during our literature review, however, there may exist classifications with aspects of performance that we have not yet considered. The set of 58 performance requirements that we used to evaluate our approach may not be large enough to draw general conclusions about the applicability.

8.5. Related Work

Incompleteness is one of the most important problems in RE leading to failed projects. In an early study, Lutz [1993] reports incompleteness as a cause of computer-related accidents and system failures. Furthermore, in a more recent study, Méndez Fernández and Wagner [2013]; Méndez Fernández and Wagner [2014] revealed in a survey with

58 industry requirements engineers, that incomplete requirements are not only named as the most frequent problem in RE, but also considered the most frequent cause for project failure. Also, Ott [2012] investigates defects in natural language requirements specifications. Their results confirm quantitatively that the most critical and important quality criteria in the investigated specifications are consistency, completeness, and correctness.

Menzel et al. [2010] report on a similar approach to ours; They propose an objective, model-based approach for measuring the completeness of functional requirements specifications. Their approach contains an information model, which formalizes the term completeness for a certain domain, a set of assignment rules, which defines how textual requirement fragments can be mapped to the information model, and a guideline, which defines how to analyze a requirements specification based on the information model. We use a similar approach, yet for the domain of performance requirements: we define a content model of performance requirements (similar to the information model) based on literature, define requirement patterns and apply the patterns to textual requirements (similar to the assignment rules and the guideline).

There is plenty of work on requirement patterns in RE. Franch et al. [2010] present a metamodel for software requirement patterns. Their approach focusses on requirement patterns as a means for reuse in different application domains and is based on the original idea of patterns by Alexander et al. [1977]. In contrast to this, the idea of our approach is to use sentence patterns for the definition of content of performance requirements in general, for the specification of performance requirements, and to define and improve the completeness of performance requirements.

Withall [2007] presents a comprehensive pattern catalogue for natural language requirements including patterns for performance requirements in his book. The pattern catalogue contains a large number of patterns for different types of requirements. In contrast to their work, our approach focusses on performance requirements and is derived step-by-step from literature. Moreover, we provide a notion of completeness for performance requirements and explicitly include the context (by means of auxiliary conditions) of performance requirements.

Filipovikj et al. [2014] conduct a case study on the applicability of requirement patterns in the automotive domain. They conclude that the concept of patterns is likely to be generally applicable for the automotive domain. In contrast to our approach, they use patterns that are intended for the real-time domain. They use Real Time Specification Pattern System as defined by Konrad and Cheng [2005] (based on the work of Dwyer et al. [1999]). These patterns use structured English grammar and support the specification of real-time properties.

Stalhane and Wien [2014] report on a case study where requirement analysts use requirement patterns to describe requirements in a structured way. Their results show that the resulting requirements are readable for humans and analyzable for their tool. Moreover, their tool improved the quality of requirements by reducing ambiguities and inconsistent use of terminology, removing redundant requirements, and improving partial and unclear requirements. In contrast to their work, we specifically focus on performance

requirements, provide a notion of completeness, and provide more detailed (and also literature-based) sentence pattern.

Wohlrab et al. [2014] present their experiences in combining existing requirements elicitation and specification methods for performance requirements. They successfully applied the so-called PROPRE method to a large industrial project and report on the lessons learnt. The PROPRE method is a comprehensive method containing various models from feature modeling to requirements templates. The method further contains requirement patterns, but on a rather abstract level. These patterns can be used for structuring information in requirements. In contrast to this, we present a step-by-step derivation and application of sentence patterns for performance requirements.

8.6. Conclusions

The goal of this chapter was to reach our third objective. In particular, our goal was to assess whether our approach is applicable in practice. To this end, we applied our approach to performance requirements and proposed a notion of completeness for performance requirements. To evaluate our approach, we conducted an empirical evaluation with respect to its applicability and ability to detect incompleteness. From the results of the study, we conclude that the proposed sentence patterns are applicable to performance requirements documented in practice. Furthermore, we argue that our approach provides a helpful and actionable definition of completeness for performance requirements that can be used to detect incompleteness and thus to pinpoint to requirements that are hard to comprehend, implement, and test. We plan to apply this approach to other quality attributes. In particular, we plan to derive a content model and a notion of completeness for other quality attributes based on literature and on the question which content is necessary to perform specific activities. This would result in activity-based definitions of quality factors, which are actionable and applicable by practitioners.

So far, our approach provides an assessment of performance requirements with respect to our notion of completeness. Considering the constructive nature of sentence patterns, if requirements are specified based on these sentence patterns, they are complete by construction. We plan to reflect this notion of completeness with the subjective assessment of practitioners and discuss whether our notion provides useful feedback. Furthermore, as requirements by means of our sentence patterns explicitly state respective functions, events, and domain objects, it would be interesting to analyze the transition to subsequent development artifacts (e.g., the architecture).

In summary, this contribution supports (the second part of) our hypothesis, i.e., a categorization based on a system model is operationalizable for subsequent development activities. In the next chapter, we summarize the contributions of this dissertation and provide an outlook on future research directions.

"Simple can be harder than complex; you have to work hard to get your thinking clean to make it simple."

— Steve Jobs

Chapter 9

Reflection on the Expressiveness of our Approach

We have proposed an approach which provides a basis for defining specific types of requirements based on a formal system model in Chapter 7 and 8. Our approach uses the system model provided by the Focus theory [Broy and Stølen, 2001] and defines the individual content elements based on the system model. Furthermore, we have instantiated the approach for performance and availability requirements and have evaluated the resulting sentence patterns.

However, there are some cases, in which the system model is not sufficient to model specific types of requirements. For example, for usability requirements; How can we define the semantics for concepts like *perceived user satisfaction*?

The purpose of this chapter is to discuss the limitations of our approach, i.e., to what degree can types of requirements be expressed based on a system model and in what cases do we need to extend the system model. Finally, we also want to discuss the considerations that justify the decision to extend—or not to extend—the system model.

9.1. Requirements, Modeling Theory, and System Model

To start our discussion, we first need to make our terminology clear. In the following, we want to make an explicit distinction between the elements of a modeling theory; We want to distinguish what we call a *behavior theory*, such as a notion of time or probabilities in the modeling theory, and the underlying *system model*, i.e., the building blocks of a system. Figure 9.1 illustrates the relationship between requirements, a modeling theory,

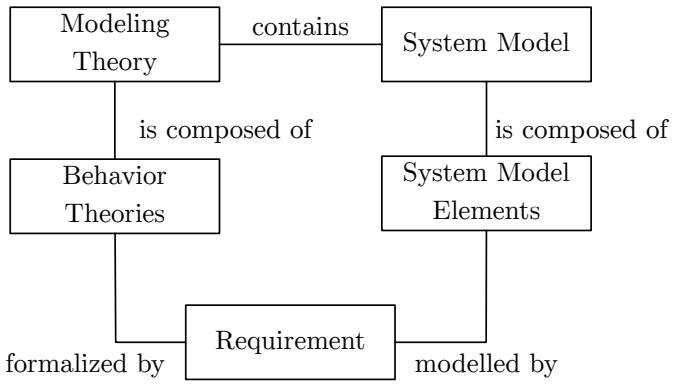


Figure 9.1.: Relationship between requirements, a modeling theory, and a system model.

and a system model. In particular, a modeling theory contains behavior theories and a system model. A behavior theory may include, for example, a notion of probability or time. A system model is composed of system model elements. Those elements define the building blocks of a system. For example, a component with a collection of input and output ports may be such a building block. A requirement is a predicate over system model elements which is formalized by a behavior theory.

The FOCUS theory [Broy and Stølen, 2001] contains a notion of probabilities [Neubeck, 2012] and a notion of time (behavior theories). Furthermore, the system model of the FOCUS theory consists of, inter alia, syntactic and semantic interfaces over typed input and output channels[16]. A requirement in the FOCUS theory is then a logical, probabilistic, timed, or timed and probabilistic predicate over the elements of the system model. Thus, in our understanding, the FOCUS theory is a modeling theory.

In the following, we discuss the relationship between types of requirements (i.e., specific subsets of requirements) and a modeling theory and its system model.

9.2. Discussion on the Relationship of Types of Requirements and Modeling Theory

In this dissertation, we have discussed two orthogonal views on requirements: (i) the view of traditional types of requirements and (ii) the view of requirements that are expressible as predicates over a system model. Figure 9.2 illustrates these two views. In the view of traditional types (left side of Figure 9.2), types of requirements are usually defined up front in a requirements categorizations, as for example the ISO/IEC 9126-2001 [2001]. They are either based on a quality model or based on a direct categorization of requirements into different classes. However, as discussed in Chapter 6, there are several problems evident with those categorizations, including a vague definition of

[16]For a detailed discussion of the system model of the FOCUS theory, we refer to Chapter 2.2 or to Broy and Stølen [2001]

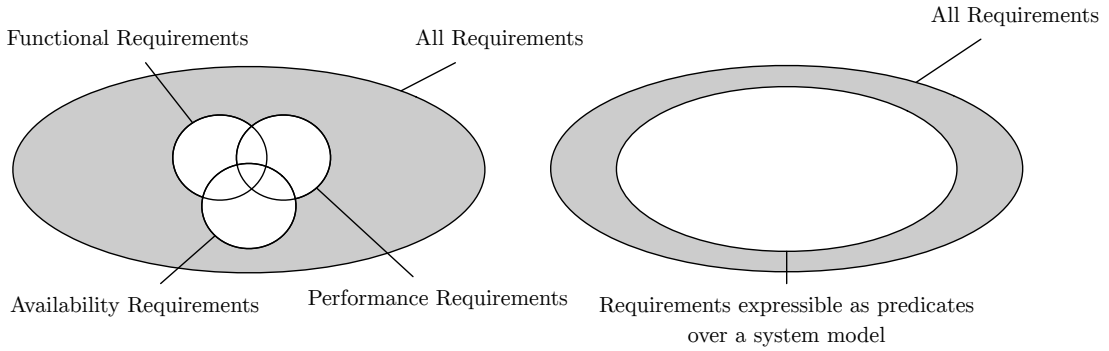


Figure 9.2.: Different views on the set of all requirements (gray). On the left: traditional types of requirements (functional, availability, and performance). On the right: requirements expressible as predicates over a system model.

the classes and missing integration in subsequent development activities. The other—orthogonal—view, we discussed in this dissertation, is to categorize requirements in all those requirements that are expressible over a system model and those that are not (right side of Figure 9.2). In contrast to the traditional view, requirements expressed based on a system model are explicitly and precisely defined and, due to the formal nature, can be integrated in subsequent development activities. For example, in requirements analysis, the requirements can be specified more precisely and in testing & validation, techniques like model checking and theorem proving (e.g. [Kurshan and Lamport, 1993; Rajan et al., 1995]) can be used. It is important to note that these requirements are restricted to behavior of a system, as we consider requirements as predicates over a system model.

According to our understanding (see Chapter 2), a functional requirement (in the traditional view) of a system expresses that (i) a system shall offer a particular functional feature such that the system can be used for a specific purpose, or (ii) a function of a system having a particular property—that may be a logical property or a probabilistic one—modeling part of the interface behavior of the system, specified by the interaction between the system and its operational context [Broy, 2015, 2016].

Thus, a functional requirement (in the traditional view) is a requirement that is expressible as a logical or probabilistic predicate over a system model (in our case over the Focus system model). Hence, functional requirements in the traditional view are contained in the set of requirements that are expressible based on the system model. Moreover, in Chapter 7, we have argued that performance requirements (in the traditional view) can be expressed as a combination of logical, timed, or probabilistic predicates over the Focus system model. Furthermore, as argued in Chapter 7 (based on the work of Junker [2016]), we can also express availability requirements (in the traditional view) as a combination of logical, timed, and probabilistic properties over the Focus system model. Hence, all these three types of requirements in the traditional view are contained in the set of requirements that are expressible based on the system model.

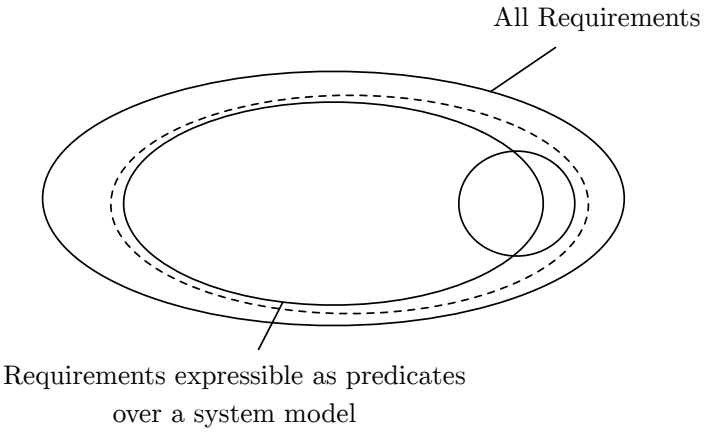


Figure 9.3.: Containment relation of all requirements and requirements that are expressible as predicated over a system model.

The main argumentation in this dissertation is that we need a modeling theory together with a system model that is expressible enough to model the types of requirements from the traditional view, i.e., requirements are expressible as predicates over the system model. Figure 9.3 illustrates the set of all requirements which is separated in the set of requirements that are expressible as predicates over a system model and the complement of this set. On the right side, a type of requirements is shown that is only partially expressible over the system model. Thus, our main question is how to choose the modeling theory and the system model such that they are expressible enough to model all relevant types of requirements? It is the goal of this chapter to discuss this question.

Note. *This question (about expressiveness of the modeling theory and the system model) is also reflected in the traditional distinction between functional and non-functional requirements. In early days of computer science (see Chapter 3.1.1 for a historical sequel of the traditional distinction), there was only a notion of logical predicates over (simple) system models, and, thus, a distinction was made between functional requirements, i.e., all those requirements that are expressible by logical predicates over a system model, and non-functional requirements, i.e., all the rest.*

Thus, the traditional distinction between functional and non-functional requirements is based on the expressiveness of a modeling theory. If the modeling theory is extended (e.g. by introducing a notion of time), we end up in a situation where a requirement that was categorized as non-functional is now categorized as functional, without changing the requirement. This situation is, in our understanding, one of the causes of the confusion around non-functional requirements.

We base the following discussion on the choice of the behavior theories and the system model on the results of Chapter 5. In particular, in Chapter 5, we analyzed 530 requirements from 11 requirements specifications from 5 different companies. We classified each of the 530 requirements in behavioral requirements, i.e., those requirements

Behavior theory	count	%
Syntactic	47	11.9%
Logical	277	69.9%
Timed	54	13.6%
Probabilistic	7	1.8%
Probabilistic & Timed	11	2.8%

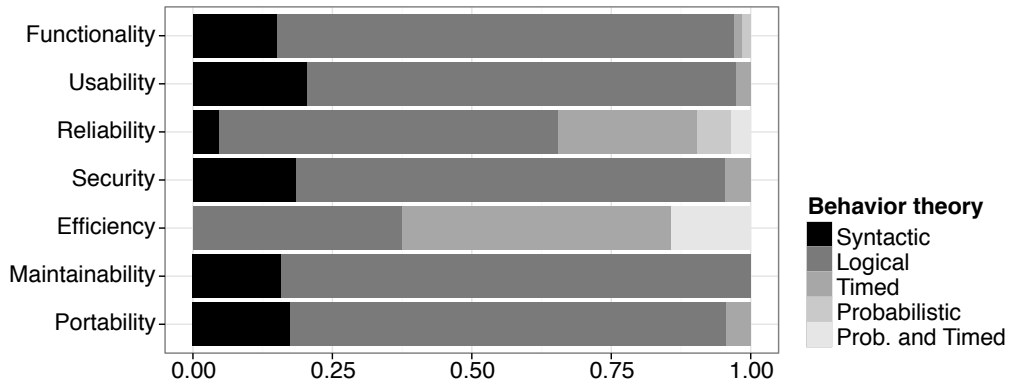


Figure 9.4.: Relative distribution of behavioral requirements with respect to their *behavior theory*: *syntactic*, *logical*, *timed*, *probabilistic*, and *probabilistic and timed* (from black to white).

that are expressible over the FOCUS system model, and representational requirements, i.e., those requirements that are *not* expressible.

9.2.1. The Choice of the Behavior Theories

We argue that it is sufficient to have logical, timed, and probabilistic predicates over a system model to describe behavioral requirements. In our document analysis in Chapter 5, we classified 74.7% of all requirements (397/530) as requirements that are expressible over the FOCUS system model. We classified these requirements according to the behavior theory that would be needed to formally specify the behavior that is expressed in the requirement. Figure 9.4 shows the results. These results show that we need a modeling theory with which we can express logical, timed, probabilistic, and probabilistic and timed requirements. The FOCUS theory [Broy and Stølen, 2001] together with its probabilistic extension [Neubeck, 2012] is a modeling theory that fulfills these requirements. Requirements can be expresses as a combination of logical, timed, and probabilistic predicates. Furthermore, the work of Broy [2015, 2016] also corroborates our hypothesis. In his work, he distinguishes between *behavioral properties* and *representational properties*. Behavioral properties are all those properties that are expressible as predicates over a system model. From the remaining 25.3% that are not expressible over the FOCUS system model, 68 are references to standards. Standard references are—in general—difficult to assess, as most standards do not provide a formal

notion of the contents of the standard. To include those requirements, we would need to model the respective standard based on the system model. Thus, we exclude references to standards in our discussion. The rest, i.e., 66 requirements, are requirements that describe the representation of the system. They cannot be expressed based on the FOCUS system model. We analyzed those requirements in detail and found that they can be formalized in a logical, timed, and probabilistic manner (based on an extended system model).

Note. *There are some cases where logical, timed, and probabilistic predicates are not enough to express behavioral requirements. For example, if we want to consider uncertainty, impreciseness, and vagueness in a qualitative fashion. For the remainder of this dissertation, we exclude those requirements. However, there is work by Koutsoumpas [2015] which integrates fuzzy concepts in the* FOCUS *theory. If we consider this extension, those requirements could also be integrated.*

9.2.2. The Choice of the System Model

The choice of the system model depends on the types of requirements we want to model, as the system model defines the basic building blocks of the system and, thus, for requirements. Hence, we can describe all those requirements that are expressible as predicates over elements of a system model. For performance requirements and availability requirements, we have already argued that the system model of the FOCUS theory is sufficient. These two types of requirements mostly describe externally observable behavior of parts of the system. However, there are some types of requirements that can not fully be modeled as predicates over the system model.

Let us consider, for example, requirements that describe the physical interaction of the system with its environment. In particular, consider an ECU that heats up the more it processes. Then, we may have a requirement like *"the system shall not heat up its environment to more than* 60 °C*"*. This requirement follows physical laws and, thus, the system model needs to contain a notion of temperature, energy, etc. In this case, we may extend the system model with a notion of temperature and temperature laws. Having this extension, we can express requirements that are formerly not expressible based on the extended system model. Note that, if we consider the traditional distinction in functional and non-functional requirements, a requirement that was perviously classified as NFR is now classified as FR, without having changed the requirement.

Another example are mechatronic systems with spatial requirements. Mechatronic systems do not only interface with other software systems or the user, but integrate electric and mechanic devices to form complex systems operating in the real world. For these systems, spatial requirements, e.g. requirements that constrain, measure, or affect the spatial relationship of physical objects, play an important role. However, the system model of the FOCUS theory only partially supports those requirements, as it, for example, does not support spatial information. For these types of systems, Hummel [2011] has extended the FOCUS system model such that positions of objects are captured over time and basic concepts required for modeling realistic systems (such as the collision between

Behavioral vs. Representational	count	%
Behavioral	396	74.7%
Representational	134	25.3%

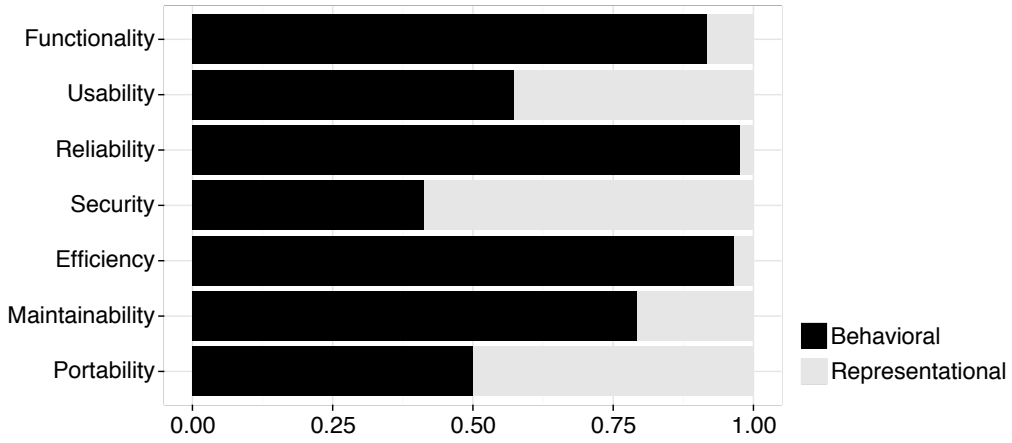


Figure 9.5.: Distribution of *behavioral* and *representational* requirements.

solid objects, the detection of objects in certain locations, material flow, and kinematic relationships between objects of the model) are supported. Again, based on this extended system model, the set of requirements that can be expressed based on the system model is enlarged.

Moreover, let us consider requirements that describe external properties in relation to the usage of the system, e.g., the usability requirement *"the time-based zoom function must be comfortable to use"*. This requirement describes an external property of the system in relation to its usage. It has a strong impact on the user interface and the interactions of the user with the software. Now we face the question how can we model the comfortability of usage in a system model? An extension of the system may be possible by integrating a notion of the user and comfortability in the system model.

To sum it up, for each type of requirement, we need to assess whether the system model is sufficient to model the kind of behavior that is described by requirements of this type. Furthermore, we need to assess—if there is a possible extension of the system model—whether it is beneficial to extend the system model.

9.2.3. Assessment of Types of Requirements w.r.t. the Focus System Model

In this section, we perform an assessment of the main quality characteristics of the ISO/IEC 9126-2001 [2001]. We base our assessment on the results of our document analysis in Chapter 5 (the results are shown in Figure 9.5). In particular, for each type of requirement, we assess for the Focus theory, if the behavior theories and system model is sufficient to express requirements of this type. Moreover, we base the assessment on the work

of Ameller et al. [2010] and Yang et al. [2014]. They conducted a literature review to find out whether current Model-Driven Development (MDD) approaches integrate QRs. They found that most current MDD approaches only focus on FRs (and thus excluding QRs). However, for our assessment we focus on those approaches that do integrate QRs into MDD according to their results. We use their results as an additional argument that specific types of requirements can be—in general—expressed based on a system model. The system models of the individual MDD approaches may be different from the system model of the FOCUS theory, however, it serves us as an indication of the feasibility.

Note. *Note that our assessment is based on the results of Chapter 5 and on reasoning. Thus, it constitutes a first step towards an unified understanding of the relationship and needs further refinement in future.*

Functionality According to ISO/IEC 9126-2001 [2001], functionality is *the capability of the software product to provide functions which meet stated and implied needs when the software is used under specified conditions.*

Assessment: As shown in Figure 9.5, we categorized 92% of the functionality requirements as requirements that are expressible over the system model. From the remaining 12 requirements, six are references to standards, like *"the back up data must be stored accordingly to [...] policies"* and six concern the syntactical or technical representation of the system. Standard references are—in general—difficult to assess, as most standards do not provide a formal notion of the contents of the standard. To include those requirements, we would need to model the respective standard based on the system model. Thus, we exclude references to standards in our discussion. Moreover, according to Ameller et al. [2010], most MDD approaches focus on functional requirements. Thus, we argue that many functional requirements from practice can be expressed as predicates over the FOCUS system model. Note that the traditional definition of functional (in contrast to non-functional) requirements is based on a simple system model. Thus, from this perspective, functional requirements should be—per definition—expressible based on a system model.

Usability According to ISO/IEC 9126-2001 [2001], usability is *the capability of the software product to be understood, learned, used and attractive to the user, when used under specified conditions.*

Assessment: In our document analysis study, we categorized 57% of all usability requirements as requirements that are expressible over the system model (as shown in Figure 9.5). We furthermore analyzed the remaining 26 requirements (excluding the three references to standards) in detail and categorized them according to the following four groups:

(i) internationalization requirements, e.g, *"[the system] must provide an English, German and French language configuration which can show 100% of [the system's] functionality"*,

(ii) accessibility requirements, e.g. *"the accessibility of the software shall be considered as far as possible"*,

(iii) common look and feel requirements, e.g. *"Graphical User Interfaces shall present a common look and feel whenever possible."*, and

(iv) UI structuring requirements, e.g., *"the description of the user rights shall be grouped according to the functionality"*.

To include those requirements, we would need to include a notion of the language of the user interface and a notion of accessibility. Furthermore, we would need to formally detail what we mean with common look and feel in the FOCUS theory. Moreover, usability requirements often constrain properties perceived by the user. To formally capture this, we would need formally specify what perceived means in the respective context.

Reliability According to ISO/IEC 9126-2001 [2001], reliability is *the capability of the software product to maintain a specified level of performance when used under specified conditions.*

Assessment: As shown in Figure 9.5, we categorized 98% of the reliability requirements as requirements that are expressible over the system model. From the remaining two requirements, one is a reference to a standard and the other one states that *"backup copies shall not be stored in the same fire area as the technical systems"*. The latter requirement can be modeled as a requirement over the extended system model by Hummel [2011], which captures positions of objects over time and basic concepts required for modeling realistic systems (such as the collision between solid objects, the detection of objects in certain locations, material flow, and kinematic relationships between objects of the model). Thus, we argue that almost all reliability requirements can be expressed as predicates over the FOCUS system model. Furthermore, as shown by Junker [2016], availability requirements—which are a subset of reliability requirements according to the ISO/IEC 9126-2001 [2001]—are expressible based on the FOCUS system model. Moreover, there are many MDD approaches [Ardagna et al., 2008; Gallotti et al., 2008; Rodrigues et al., 2005; Wada et al., 2010] that integrate reliability requirements. This further yields an argument, that a large portion of the reliability requirements can be expressed based on a system model.

Security According to ISO/IEC 9126-2001 [2001], security is *the capability of the software product to protect information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them.*

Assessment: In our document analysis study, we categorized 41% of all security requirements as requirements that are expressible over the system model (as shown in Figure 9.5). However, many of the security requirements we analyzed were references to standards. If we exclude those from our analysis, 93% are expressible as predicates over the system model. The remaining 3 requirements constraint

the syntactical representation of the system. Furthermore, there are some MDD approaches [Wada et al., 2010] that integrate security requirements based on, e.g, an UML profile. Thus, we argue that many security requirements from practice can be expressed as predicates over the FOCUS system model. However, we would need a more detailed analysis for security requirements.

Efficiency According to ISO/IEC 9126-2001 [2001], efficiency is *the capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions.*
Assessment: As shown in Figure 9.5, we categorized 97% of the efficiency requirements as requirements that are expressible over the system model. This is in tune with our results in Chapter 7 and 8. The remaining two requirements are requirements that concern the usage of specific hardware, e.g., *"the system shall support and leverage 64-bit hardware and operating systems for scaling up"*. To include these types of requirements in the FOCUS theory, we would need a notion of the executing hardware. Furthermore, there are some MDD approaches [Ardagna et al., 2008; Gallotti et al., 2008; Gönczy et al., 2009; Kugele et al., 2008] that integrate efficiency requirements. Thus, we argue that almost all efficiency requirements can be expressed as predicates over the FOCUS system model.

Maintainability According to ISO/IEC 9126-2001 [2001], maintainability is *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.*
Assessment: As shown in Figure 9.5, we categorized 79% of the maintainability requirements as requirements that are expressible over the system model. We categorized the remaining requirements in the following two groups:

(i) requirements that concern the artifact documentation, e.g. *"the system shall be sufficiently documented. This concerns the inline comments in the source code"* and

(ii) technology requirements, e.g., *"the used technologies for hardware and software shall not lead to a vendor lock in"*.

To formally specify requirements that concern the artifact documentation, we would need to extend the system model with a notion of artifacts. Similarly for technology requirements; These would need a notion of used technologies for software and hardware. Furthermore, the literature review of Ameller et al. [2010] did not yield any results concerning MDD approaches that integrate maintainability requirements. Still, we argue that a large portion of maintainability requirements can be expressed as predicates over the FOCUS system model. For the remaining requirements, we would need to extend the system model.

Portability According to ISO/IEC 9126-2001 [2001], portability is *the capability of the software product to be transferred from one environment to another.*

Assessment: Finally, as shown in Figure 9.5, we categorized 50% of the portability requirements as requirements that are expressible over the system model. If we analyze the remaining 23 requirements in detail, we can categorize them in

(i) hardware requirements like *"the system platform for the [system] shall be [platform]"*,

(ii) infrastructure/environment requirements like *"the system shall support deployment in a virtualized environment such as [environment]"*, and

(iii) client profile requirements like *"[the system] must support retailers who sell several brands (both of them)"*.

To include those requirements, we would need to extend the FOCUS theory to include a notion of specific details of hardware, infrastructure, and environment and about client profiles. Furthermore, the literature review of Ameller et al. [2010] did not yield any results concerning MDD approaches that integrate portability requirements. Still, a large portion of portability requirements can be expressed as predicates over the system model.

The question when to extend the system model can be broken down to the question how beneficial would it be to have formal analyses for a given type of requirements. For example, in requirements analysis, the requirements can be specified more precisely based on the system model and in testing, techniques like model checking and theorem proving (e.g. [Kurshan and Lamport, 1993; Rajan et al., 1995]) can be used to show that requirements are fulfilled by the system. Thus, if we need to conduct formal analyses of our requirements, as for example required by the ISO/IEC 26262-2011 [2011] for all requirements that are relevant for safety, an extension of the system model is beneficial.

9.3. Conclusion & Future Work

In this chapter, we have discussed the limitations of our approach. In particular, we have discussed the influence of the choice of the behavioral theories and system model towards categorizations of requirements. Moreover, we discussed further types of requirements with respect to their expressiveness in the FOCUS theory.

We argue that the choice of the modeling theory and, in particular, the system model strongly influences what types of requirements we can modeled. Requirements that are formally modeled can subsequently be formally analyzed. We argue that it is sufficient to include a notion of time and probability in a modeling theory. Furthermore, we discussed that the system model of the FOCUS theory is expressive enough to model a large amount of requirements. In particular, functional, reliability, security, and efficiency requirements can be expressed to a large amount based on the FOCUS system model. For usability, maintainability, portability requirements, the system model needs to be extended.

In future, we propose to further refine the assessment of types of requirements. In particular, we propose to further analyze what subset of requirements can be expressed based on a system model and to what degree is it justified to extend the system model.

Thus, for the FOCUS theory, the question is where are the limits of its expressiveness (w.r.t. requirements) and to what degree is it justified to extend its system model.

"No book can ever be finished. While working on it we learn just enough to find it immature the moment we turn away from it."

— Karl Popper

10 Chapter
Conclusions & Outlook

The goal of this final chapter is to summarize the main contributions of this dissertation, conclude the dissertation, and discuss possible directions for future research.

10.1. Summary of Conclusions

This dissertation is built on the hypothesis that *a requirements categorization based on a system model is adequate for requirements found in practice and is operationalizable for subsequent development activities.* In this dissertation, with *adequacy of a requirements categorization*, we mean that the categorization is applicable for industrial requirements and, furthermore, supports subsequent development activities. We argued that such a categorization provides us with a clear notion and concept of a system. Based on the categorization, we can precisely specify requirements in terms of properties of systems, where properties are represented by logical predicates. This allows us to precisely and explicitly specify the structuring principles of the categorization. Given a requirements categorization that is based on a system model, the seamless transition to architectural design (operationalization) is facilitated, as requirements are built on clearly defined and explicitly stated logical properties over a set of systems.

We claimed that this dissertation provides supporting evidence and solutions for the stated hypothesis. In the following, we conclude that these claims are supported by the contributions provided in this thesis.

Adequacy of a Requirements Categorization based on a System Model

In Chapter 5, we analyzed whether a requirements categorization based on a system model is adequate for requirements found in practice. In particular, we analyzed 11 requirements specifications from 5 different companies working in different application domains and of different sizes. We collected all those requirements that are labeled as "non-functional", "quality", or any specific quality attribute, resulting in 530 requirements in total.

The results of this study show that 75% of the requirements labeled as "quality" in the considered industrial specifications describe system behavior and 25% describe the representation of the system. From the QRs that describe system behavior, 69% describe behavior over the interface of the system, 21% describe architectural behavior and 10% describe state behavior. We furthermore discussed the implications we see on handling QRs in the software development phases, e.g., testing or design.

Based on these results, we argue that functional requirements describe any kind of behavior over the interface of the system, including timing and/or probabilistic behavior. From this perspective, we conclude that many of those QRs that address system properties describe the same type of behavior as functional requirements do (see column *Interface* in Figure 5.6). This is true for almost all QR classes we analyzed; even for QR classes which are sometimes called *internal* quality attributes (e.g., portability or maintainability) [McConnell, 2004]. Hence, we argue that Broy's requirements categorization—that is based on a system model—is adequate for requirements found in practice, as the categories can be linked to system development activities. From a practical point of view, this means that most QRs can be elicited, specified, and analyzed like functional requirements. For example, QRs classified as black-box interface requirements, are candidates for system tests. In our data set, system test cases could have been specified for almost 51.5% of the QRs.

Moreover, in Chapter 9, we discussed the limitations of modeling requirements based on a system model. We conclude that it is sufficient to include a notion of time and probability in a modeling theory. Furthermore, we discussed that the system model of the Focus theory is expressive enough for relevant types of requirements. In particular, functional, performance, reliability, safety, and security requirements can be expressed based on the Focus system model. For usability, maintainability, reusability, releasability, and supportability requirements, the system model is not sufficient. However, we believe that an extension of the system model is—in general—not justified, as those types of requirements are usually not formally specified/documented and there are usually no formal analyses conduced with those requirements in subsequent development activities.

This contribution supports (the first part of) our hypothesis, i.e., a categorization based on a system model is adequate for requirements found in practice.

Operationalization of a Requirements Categorization based on a System Model

In Chapter 7, we presented an approach for defining, specifying, and integrating QRs based on a system model. In particular, the approach takes a specific quality attribute as input and creates a precise and explicit definition and customized sentence patterns for requirements concerning this quality attribute. The resulting definitions and sentence patterns can then be integrated in the overall RE process to support the documentation, elicitation, management, and validation of requirements in the given organizational context. We furthermore instantiated our approach for the quality attributes performance and availability and provided a discussion of our lessons learnt while instantiating it. In particular, we presented a context-independent and context-dependent content model for performance requirements, an (informal) definition of the content elements and a discussion on how to express them based on the Focus system model, and an operationalization through sentence patterns for the specification of performance requirement. Furthermore, we derived a notion of completeness for performance requirements based on the context-dependent content model.

Furthermore, in Chapter 8, we conducted an evaluation of our approach for performance requirements. In particular, we applied the resulting sentence patterns for performance requirements to 58 performance requirements taken from 11 industrial specifications and analyzed (i) the applicability and (ii) the ability to uncover incompleteness of performance requirements.

We were able to rephrase 86% of the performance requirements. Moreover, we found that the resulting sentence patterns can be used to detect incompleteness in performance requirements, revealing that 68% of the analyzed performance requirements were incomplete.

In summary, we conclude that our approach is applicable for performance and availability requirements and its results can be used to support the documentation, elicitation, management, and validation of requirements in the given organizational context. Thus, we argue that our approach provides an operationalization for subsequent development activities of a requirements categorization based on a system model.

10.2. Overall Conclusion & Implications

In this dissertation, we have analyzed the state of the practice of how practitioners handle requirements. Based on this, we analyzed the adequacy of a requirements categorization based on a system model with 530 requirements from industrial specifications and concluded that the categorization is adequate. With *adequacy of a requirements categorization*, we mean that the categorization is applicable for industrial requirements and, furthermore, supports subsequent development activities. Furthermore, we presented an approach for defining, specifying, and integrating QRs based on a system model and conducted an evaluation with respect to its applicability and ability to uncover incompleteness.

In summary, we conclude that a requirements categorization based on a system model is adequate for industrial requirements and, furthermore, that it can be operationalized for subsequent development activities. Thus, we conclude that we can support our hypothesis.

Personally, we are convinced that the purpose of a categorization (in general) is to clearly and unambiguously categorize elements in categories according to clearly defined arguments. Furthermore, and even more important, a categorization should have a clearly defined purpose. For requirements categorizations, this means that on the one hand—from an academic perspective—a categorization should clearly and unambiguously categorize requirements. On the other hand—from a practical perspective—a categorization should categorize requirements in a way such that the activities that are performed with the requirements can be aligned according to the categories.

Following these goals, we argue that we should reconsider categorizing requirements simply into functional and non-functional requirements. This simple categorization neither unambiguously categorizes requirements (see for example the discussion of Glinz [2007] or Broy [2016]) nor sufficiently supports subsequent development activities. This is in line with the established opinion in the field, e.g. with Pohl [2010] and Glinz [2007].

With respect to requirements categorizations that go beyond a categorization in functional and non-functional, we found in our initial literature review that they more or less clearly and unambiguously categorize elements according to clearly defined arguments. One example is the concern-based classification of Glinz [2007] and Broy's categorization that is based on a system model. However, concerning the applicability in practice, we found in the first part of this dissertation that there are several problems evident with current categorizations, as for example that traceability becomes expensive or that QRs are even forgotten (see Chapter 4). Furthermore, existing literature (e.g., [Ameller et al., 2012; Borg et al., 2003; Chung and Nixon, 1995; Svensson et al., 2009]) indicates that QRs are not sufficiently integrated in the overall development process. This is where the second part of this dissertation steps in. Our approach (see Chapter 7) is an extension of a given categorization, which clearly defines the individual categories and thus reduces ambiguity. For example, we provided a content model for performance requirements including a clear definition of the individual content elements in Section 7.4. Furthermore, we provide a means, i.e., sentence patterns, for the specification of requirements concerning the individual categories and thus support subsequent development activities.

From a researchers' perspective, the results of this dissertation strengthen our confidence that current requirements categorizations need to be extended to meet these two goals. Our approach is a step towards a clear understanding of the categories and the application of requirements categorizations.

From a practitioners' perspective, not only the operationalization via sentence patterns could be easily implemented in a requirements authoring or management tool. Such a tool may provide instant feedback to the requirements engineer about missing or optional content elements, similar to requirements smells [Femmer et al., 2014a,b, 2016; Vogelsang et al., 2016]. Furthermore, the tool might check the terms used in a requirement with

respect to an underlying domain model and then uncover terms that are neither part of the consolidated terminology nor defined through the pattern semantics.

A long-term vision that emerges from our results is that we might be able to better integrate requirements categorizations into a holistic software and system development process in the future. Such an integration would yield, for instance, seamless modeling of all properties associated with a system. The benefits of such an integration include that specific categories would not be neglected during development activities, as it is too often current state of practice; from an improvement in the traceability of requirements over an improvement of possibilities for progress control to an improvement of validation and verification.

10.3. Outlook

In this section, we describe possible future research directions.

10.3.1. Application of the Approach to further Quality Attributes

In this dissertation, we proposed an approach for defining, specifying, and integrating QRs. We instantiated our approach for two specific quality attributes (performance and availability) and conducted an empirical evaluation with respect to its applicability. The results indicate that the approach is applicable and besides the constructive nature of our approach, further supports analytical quality assessment with syntactic analyses. For example, the question how can we assess that all information necessary for the subsequent development activities are documented in a given textual requirement (i.e., the completeness of the individual requirement)?

Our approach captures the content of a requirement as a model. Building such models for industrial requirements allows reasoning about several statements that are presumed to be common knowledge about QRs. For example, the assertion that QRs are cross-functional and affect the whole system is challenged by the fact that a reasonable share of examined requirements regarded the scope “function” or “component” instead of “system”.

We consider the (re)definition of individual quality attributes, as we did with performance and availability in this paper, based on their impact to development activities as beneficial for the definition of quality attributes. Thus, our approach could be applied to other quality attributes, resulting in a content model and a notion of completeness for other quality attributes based on literature and on the question which content is necessary to perform specific activities. This would result in activity-based definitions of quality factors, which are actionable and applicable by practitioners.

As a long term vision, the resulting content models could be analyzed with respect to their commonalities and eventually unified in one content model for quality requirements (for all quality attributes). This would result in a model of all content elements that characterize the specification of quality requirements. Based on this, we could give a precise and explicit definition of quality requirements and, furthermore, an integration in

the development process by means of the respective sentence patterns. Given this unified content model, further questions arise like its applicability, usefulness, and adequacy.

10.3.2. Integration in Analytical Quality Assessments

The results of the instantiation of our approach to performance requirements suggest that natural language performance requirements in practice are, to a large extent, incomplete with respect to our notion of completeness or at least need to be interpreted to be implemented and tested. Our approach is a step towards increasing the completeness of quality requirements in general. The operationalization via sentence patterns could be easily implemented in a requirements authoring or management tool. Such a tool may provide instant feedback to the requirements engineer about missing or optional content elements. Furthermore, the tool might check the terms used in a requirement with respect to an underlying domain model to uncover terms, the reader must interpret because the term is not part of the consolidated terminology. Furthermore, it would be interesting to integrate the notion of completeness that is associated with our approach with requirements smells [Femmer et al., 2014a,b, 2016; Vogelsang et al., 2016], i.e., we could automatically analyze requirements documents and provide an indication for a quality defect if requirements are incomplete with respect to our notion.

An additional benefit of our approach is that it makes content in natural language requirements explicit and traceable through content elements. This allows connecting specific content elements of requirements with specific content elements in related artifacts such as test cases or components within the implementation. Updates within requirements may then be propagated directly to corresponding test cases for example, making maintenance activities more efficient and effective.

10.3.3. Evaluation of the Cost and Benefits of our Approach

So far, our approach provides a means for practitioners to specify requirements concerning a specific quality attribute. Furthermore, it provides an assessment of quality requirements with respect to our notion of completeness. Considering the constructive nature of sentence patterns, if requirements are specified based on these sentence patterns, they are complete by construction. However, conducting our approach for a quality attribute is labor intensive. Thus, the question is about the cost benefit ratio; Building a cost model for our approach and connecting this cost model with the associated benefits is a challenging task. The results will help to better understand when to apply the approach and when not to apply it.

10.3.4. Integration of our Approach in a Specification Methodology for Requirements

In Chapter 8, we have seen that in contrast to the prevailing opinion that QRs are cross-functional, the scope of only 58% is the whole system, for 34% it is a function and for 8% a component. For time behavior requirements, the percentage of requirements having a function as scope (49%) even rules out the percentage of requirements having

the system as scope (46%). In contrast to this, for capacity requirements, 85% of the requirements specify the system as scope and only 15% a component as scope.

These results lead to an interesting question: How can we integrate our approach in a requirements specification methodology. For example, if we consider the requirements that have a function as scope. Can we integrate them with a use case-based methodology, i.e. to structure the functionality and also the QRs of a system by functions [Broy, 2010b; Vogelsang, 2015; Vogelsang et al., 2015]?

10.3.5. Application Beyond Quality Requirements

So far, we have set the scope to product-related requirements and traditional quality requirements according to the ISO/IEC 25010-2011 [2011]. Thus, we explicitly excluded process-related requirements and also requirements that concern physical aspects of a system, like spatial requirement, i.e., requirements that describe properties about the spatial relationship of a system with physical objects. As an interesting open question, we propose to apply our approach also to these types of requirements, resulting in a precise definition and an operationalization by means of sentence patterns. For example, if we apply our approach to process-related requirements, it would result in a model of content elements that are relevant to specify these kind of requirements. In contrast to this work, a process model could then be used to precisely define the content elements. It would be interesting to analyze the relationships between these requirements and functional/quality requirements by leveraging the connection between system models and process models. If we are able to unify these models, we would enable a requirements engineer to specify quality requirements, physical requirements, and functional requirements based on one unified method.

"A man will turn over half a library to make one book."
— Samuel Johnson

Bibliography

Adolph, S., Hall, W., and Kruchten, P. (2011). Using Grounded Theory to Study the Experience of Software Development. *Empirical Software Engineering*, 16:487–513.

Al-Qutaish, R. E. (2010). Quality Models in Software Engineering Literature: An Analytical and Comparative Study. *Journal of American Science*, 6(3):166–175.

Alexander, C. (1979). *The Timeless Way of Building*. Oxford Books.

Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

Ameller, D., Ayala, C., Cabot, J., and Franch, X. (2012). How do Software Architects Consider Non-functional Requirements: An Exploratory Study. In *Proceedings of the 20th International Requirements Engineering Conference (RE)*, pages 41–50.

Ameller, D., Franch, X., and Cabot, J. (2010). Dealing with Non-Functional Requirements in Model-Driven Development. In *Proceedings of the 18th International Requirements Engineering Conference (RE)*, pages 189–198.

Antón, A. I. (1997). *Goal Identification and Refinement in the Specification of Software-based Information Systems*. PhD thesis, Georgia Institute of Technology.

Ardagna, D., Ghezzi, C., and Mirandola, R. (2008). Rethinking the Use of Models in Software Architecture. In *Proceedings of the 4th International Conference on the Quality of Software-Architectures (QoSA)*, pages 1–27.

Balsamo, S., Di Marco, A., Inverardi, P., and Simeoni, M. (2004). Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310.

Bauer, F. L., Bolliet, L., and Helms, H. (1968). Report on a Conference Sponsored by the NATO Science Committee. In *NATO Software Engineering Conference*.

Bauer, H. (2001). *Measure and Integration theory*. de Gruyter.

Behkamal, B., Kahani, M., and Akbari, M. K. (2009). Customizing ISO 9126 quality model for evaluation of B2B applications. *Information and software technology*, 51(3).

Bianchi, G. (2000). Performance Analysis of the IEEE 802.11 Distributed Coordination Function. *IEEE Journal on selected areas in communications*, 18(3):535–547.

Boehm, B. W. (1976). *Software Engineering*. IEEE Transactions on Computers.

Boehm, B. W. (1981). *Software Engineering Economics*. Prentice Hall PTR.

Boehm, B. W., Brown, J. R., and Lipow, M. (1976). Quantitative Evaluation of Software Quality. In *Proceedings of the 2nd international conference on Software engineering (ICSE)*, pages 592–605.

Börcsök, J. (2011). *Funktionale Sicherheit – Grundzüge sicherheitstechnischer Systeme*. VDE Verlag, 3 edition.

Borg, A., Yong, A., Carlshamre, P., and Sandahl, K. (2003). The Bad Conscience of Requirements Engineering: An Investigation in Real-world Treatment of Non-functional Requirements. In *Proceedings of the 3rd Conference on Software Engineering Research and Practice in Sweden (SERPS)*, pages 1–8.

Botella, P., Burgués, X., Carvallo, J., Franch, X., Grau, G., Marco, J., and Quer, C. (2004). ISO/IEC 9126 in practice: what do we need to know. In *Proceedings of the 1st Software Measurement European Forum (SMEF)*, pages 297–306.

Broy, M. (1998). A Functional Rephrasing of the Assumption/Commitment Specification Style. *Formal Methods in System Design*, 13(1):87–119.

Broy, M. (2010a). A Logical Basis for Component-Oriented Software and Systems Engineering. *The Computer Journal*, 53(10):1758–1782.

Broy, M. (2010b). Multifunctional software systems: Structured modeling and specification of functional requirements. *Science of Computer Programming*, 75(12):1193–1214.

Broy, M. (2015). Rethinking Nonfunctional Software Requirements. *IEEE Computer*, 48(5):96–99.

Broy, M. (2016). Rethinking Nonfunctional Software Requirements: A Novel Approach Categorizing System and Software Requirements. In Hinchey, M., editor, *Software Technology: 10 Years of Innovation in IEEE Computer*. John Wiley & Sons/IEEE Press.

Broy, M., Chakraborty, S., Goswami, D., Ramesh, S., Satpathy, M., Resmerita, S., and Pree, W. (2011). Cross-layer Analysis, Testing and Verification of Automotive Control Software. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 263–272.

Broy, M., Krüger, I. H., and Meisinger, M. (2007). A Formal Model of Services. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(1).

Broy, M. and Stølen, K. (2001). *Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement*. Springer.

Chen, L., Ali Babar, M., and Nuseibeh, B. (2013). Characterizing Architecturally Significant Requirements. *IEEE Software*, 30(2):38–45.

Chung, L. and do Prado Leite, J. C. S. (2009). On Non-Functional Requirements in Software Engineering. In *Conceptual modeling: Foundations and applications*, volume 5600 of *Lecture Notes in Computer Science*, pages 363–379. Springer.

Chung, L. and Nixon, B. A. (1995). Dealing with Non-Functional Requirements: Three Experimental Studies of a Process-Oriented Approach. In *Proceedings of the 17th International Conference on Software Engineering (ICSE)*, pages 25–37.

Chung, L., Nixon, B. A., Yu, E., and Mylopoulos, J. (2012). *Non-Functional Requirements in Software Engineering*, volume 5. Springer Science & Business Media.

Cleland-Huang, J., Hanmer, R. S., Supakkul, S., and Mirakhorli, M. (2013). The Twin Peaks of Requirements and Architecture. *IEEE Software*, 30(2):24–29.

Dahl, O.-J., Dijkstra, E. W., and Hoare, C. A. R. (1972). *Structured programming*. Academic Press Ltd.

Damm, W., Votintseva, A., Metzner, A., Josko, B., Peikenkamp, T., and Böde, E. (2005). Boosting Re-use of Embedded Automotive Applications through Rich Components. In *Proceedings of the Workshop on Foundations of Interface Technologies (FIT)*.

Davis, A. M. (1993). *Software Requirements: Objects, Functions, and States*. Prentice-Hall, Inc.

de Almeida Ferreira, D. and Rodrigues da Silva, A. (2013). RSL-PL: A Linguistic Pattern Language for Documenting Software Requirements. In *3rd International Workshop on Requirements Patterns (RePa)*, pages 17–24.

Deissenboeck, F. (2009). *Continuous Quality Control of Long-Lived Software Systems*. PhD thesis, Technische Universität München.

Deissenboeck, F., Wagner, S., Pizka, M., Teuchert, S., and Girard, J. (2007). An Activity-Based Quality Model for Maintainability. In *Proceedings of the 23rd International Conference on Software Maintenance (ICSM)*, pages 184–193.

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *Numerical Mathematic*, 1(1):269–271.

Dijkstra, E. W. (1968). Letters to the Editor: Go to Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148.

Dijkstra, E. W. (1972). The Humble Programmer. *Communications of the ACM*, 15(10):859–866.

Dromey, G. R. (1995). A Model for Software Product Quality. *IEEE Transactions on Software Engineering*, 21(2):146–162.

Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1999). Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, pages 411–420.

Eckhardt, J., Méndez Fernández, D., and Vogelsang, A. (2015). How to specify Non-functional Requirements to support seamless modeling? A Study Design and Preliminary Results. In *Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 164–167.

Eckhardt, J., Vogelsang, A., and Femmer, H. (2016a). An Approach for Creating Sentence Patterns for Quality Requirements. In *Proceedings of the 6th International Workshop on Requirements Patterns (RePa)*, pages 308–315.

Eckhardt, J., Vogelsang, A., Femmer, H., and Mager, P. (2016b). Challenging Incompleteness of Performance Requirements by Sentence Patterns. In *Proceedings of the 24th International Requirements Engineering Conference (RE)*, pages 46–55.

Eckhardt, J., Vogelsang, A., and Méndez Fernández, D. (2016c). Are Non-functional Requirements Really Non-functional? An Investigation of Non-functional Requirements in Practice. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 832–842.

Eckhardt, J., Vogelsang, A., and Mendéz Fernández, D. (2016d). On the Distinction of Functional and Quality Requirements in Practice. In *Proceedings of the 17th International Conference on Product-Focused Software Process Improvement (PROFES)*, pages 31–47.

Eder, S., Femmer, H., Hauptmann, B., and Junker, M. (2014). Which features do my users (not) use? In *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 446–450.

Fatwanto, A. and Boughton, C. (2008). Analysis, Specification and Modeling of Non-Functional Requirements for Translative Model-Driven Development. In *Proceedings of the 4th International Conference on Computational Intelligence and Security (CIS)*, pages 405–410.

Femmer, H., Kučera, J., and Vetrò, A. (2014a). On the Impact of Passive Voice Requirements on Domain Modelling. In *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 211–214.

Femmer, H., Méndez Fernández, D., Juergens, E., Klose, M., Zimmer, I., and Zimmer, J. (2014b). Rapid Requirements Checks with Requirements Smells: Two Case Studies. In *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering (RCoSE)*, pages 10–19.

Femmer, H., Méndez Fernández, D., Wagner, S., and Eder, S. (2016). Rapid quality assurance with Requirements Smells. *Journal of Systems and Software (JSS)*.

Femmer, H., Mund, J., and Méndez Fernández, D. (2015). It's the Activities, Stupid!: A New Perspective on RE Quality. In *Proceedings of the 2nd International Workshop on Requirements Engineering and Testing (RET)*, pages 13–19.

Filipovikj, P., Nyberg, M., and Rodriguez-Navas, G. (2014). Reassessing the Pattern-Based Approach for Formalizing Requirements in the Automotive Domain. In *Proceedings of the 22nd International Requirements Engineering Conference (RE)*, pages 444–450.

Floyd, R. W. (1967). Assigning Meanings to Programs. *Mathematical aspects of computer science*, 19(19-32):65–81.

Franch, X., Palomares, C., Quer, C., Renault, S., and De Lazzer, F. (2010). A Metamodel for Software Requirement Patterns. In *Proceedings of the 16th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*, pages 85–90. Springer.

Franch, X., Quer, C., Renault, S., Guerlain, C., and Palomares, C. (2013). Constructing and Using Software Requirement Patterns. In *Managing Requirements Knowledge*, pages 95–116. Springer.

Gallotti, S., Ghezzi, C., Mirandola, R., and Tamburrelli, G. (2008). Quality Prediction of Service Compositions through Probabilistic Model Checking. In *Proceedings of the 4th International Conference on the Quality of Software-Architectures (QoSA)*, pages 119–134.

Glinz, M. (2005). Rethinking the Notion of Non-Functional Requirements. In *Proceedings of the 3rd World Congress for Software Quality (WCSQ)*, pages 55–64.

Glinz, M. (2007). On Non-Functional Requirements. In *Proceedings of the 15th International Requirements Engineering Conference (RE)*, pages 21–26.

Glinz, M. (2014). A Glossary of Requirements Engineering Terminology. *Standard Glossary of the Certified Professional for Requirements Engineering (CPRE) Studies and Exam*, Version 1.6.

Gönczy, L., Déri, Z., and Varró, D. (2009). Model Transformations for Performability Analysis of Service Configurations. In *Models in Software Engineering*, volume 5421 of *Lecture Notes in Computer Science*, pages 153–166. Springer.

Grady, R. B. (1992). *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, Inc.

Gray, R. M., Dubois, E., Gray, J., Adm, R., Gray, A. H., and Dubois, S. J. (2001). *Probability, Random Processes, and Ergodic Properties*. Springer.

Hoare, C. A. R. (1969). An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580.

Hummel, B. (2011). *Integrated Behavior Modeling of Space-Intensive Mechatronic Systems*. PhD thesis, Technische Universität München.

IEEE Std 610.12-1990 (1990). IEEE Standard Glossary of Software Engineering Terminology—IEEE Std 610.12-1990. IEEE Standard, Software Engineering Standards Committee and IEEE-SA Standards Board.

IEEE Std 830-1998 (1998). IEEE Recommended Practice for Software Requirements Specifications—IEEE Std 830-1998. IEEE Standard, Software Engineering Standards Committee and IEEE-SA Standards Board.

ISO/IEC 15288-2008 (2008). International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). ISO/IEC 15288: Systems and software engineering – System life cycle processes.

ISO/IEC 25010-2011 (2011). International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). ISO/IEC 25010: Systems and software engineering–Systems and software Quality Requirements and Evaluation (SQuaRE)–System and software quality models.

ISO/IEC 26262-2011 (2011). International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). ISO/IEC 26262: Road vehicles – Functional safety.

ISO/IEC 9000-2000 (2000). International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). ISO/IEC 9000: Quality Management Systems–Fundamentals and Vocabulary.

ISO/IEC 9126-2001 (2001). International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC). ISO/IEC 9126-1: Software Engineering-Product Quality-Part 1: Quality Model.

ISO/IEC/IEEE 29148-2011 (2011). International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) and IEEE. ISO/IEC/IEEE 29148: Systems and software engineering – Life cycle processes – Requirements engineering.

Jacobson, I., Booch, G., Rumbaugh, J., Rumbaugh, J., and Booch, G. (1999). *The Unified Software Development Process*, volume 1. Addison-Wesley Reading.

Jain, R. (1990). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley & Sons.

Jansen, M. G. and Prasad, R. (1994). Capacity, Throughput, and Delay Analysis of a Cellular DS CDMA System with Imperfect Power Control and Imperfect Sectorization. *IEEE Trans. on Vehicular Technology.*

Junker, M. (2016). *Specification and Analysis of Availability for Software-Intensive Systems.* PhD thesis, Technische Universität München.

Junker, M. and Neubeck, P. (2012). A Rigorous Approach to Availability Modeling. In *Proceedings of the Workshop of Modeling in Software Engineering (MISE)*, pages 1–7.

Kitchenham, B. A. and Pfleeger, S. L. (2008). Personal Opinion Surveys. In *Guide to Advanced Empirical Software Engineering*, pages 63–92. Springer.

Konrad, S. and Cheng, B. H. C. (2005). Real-time specification patterns. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 372–381.

Kopczyńska, S. and Nawrocki, J. (2014). Using Non-functional Requirements Templates for Elicitation: A Case Study. In *Proceedings of the 4th International Workshop on Requirements Patterns (RePa)*, pages 47–54.

Koutsoumpas, V. (2015). A Formal Approach based on Fuzzy Logic for the Specification of Component-Based Interactive Systems. In *Proceedings 12th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2015)*, pages 62–76.

Kugele, S., Haberl, W., Tautschnig, M., and Wechs, M. (2008). Optimizing Automatic Deployment Using Non-functional Requirement Annotations. In *Leveraging Applications of Formal Methods, Verification and Validation*, volume 17 of *Communications in Computer and Information Science*, pages 400–414. Springer Berlin Heidelberg.

Kurshan, R. P. and Lamport, L. (1993). Verification of a multiplier: 64 bits and beyond. In *Proceedings on the 5th International Conference on Computer Aided Verification (CAV)*, pages 166–179. Springer Berlin Heidelberg.

Landes, D. and Studer, R. (1995). *The Treatment of Non-Functional Requirements in MIKE.* Springer.

Lauesen, S. (2002). *Software Requirements: Styles and Techniques.* Pearson Education.

Lochmann, K. and Wagner, S. (2012). A Quality Model for Software Quality. Technical report, Technische Universität München. Technical Report.

Lutz, R. R. (1993). Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems. In *Proceedings of the 3rd International Symposium on Requirements Engineering (RE)*, pages 126–133.

Mager, P. (2015). Towards a Profound Understanding of Non-Functional Requirements. Master's thesis, Technische Universität München.

Mairiza, D., Zowghi, D., and Nurmuliani, N. (2010). An Investigation into the Notion of Non-Functional Requirements. In *Proceedings of the 25th ACM Symposium on Applied Computing (SAC)*, pages 311–317.

McCall, J. A., Richards, P. K., and Walters, G. F. (1977). Factors in software quality. Concepts and Definitions of Software Quality. Technical report, General Electric Company. Technical Report.

McCarthy, J. (1961). A basis for a mathematical theory of computation, preliminary report. In *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference*, pages 225–238. ACM.

McConnell, S. (2004). *Code Complete*. Pearson Education.

Méndez Fernández, D. (2011). *Requirements Engineering: Artefact-Based Customisation*. PhD thesis, Technische Universität München.

Méndez Fernández, D. and Penzenstadler, B. (2015). Artefact-based Requirements Engineering: The AMDiRE Approach. *Requirements Engineering*, 20(4):405–434.

Méndez Fernández, D., Penzenstadler, B., Kuhrmann, M., and Broy, M. (2010). A Meta Model for Artefact-orientation: Fundamentals and Lessons Learned in Requirements Engineering. In *Model driven engineering languages and systems (MoDELS)*, pages 183–197. Springer.

Méndez Fernández, D. and Wagner, S. (2013). Naming the Pain in Requirements Engineering: Design of a Global Family of Surveys and First Results from Germany. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 183–194.

Méndez Fernández, D. and Wagner, S. (2014). Naming the Pain in Requirements Engineering: A Design for a Global Family of Surveys and First Results from Germany. *Information and Software Technology*, 57:616–643.

Menzel, I., Mueller, M., Gross, A., and Doerr, J. (2010). An Experimental Comparison Regarding the Completeness of Functional Requirements Specifications. In *Proceedings of the 18th International Requirements Engineering Conference (RE)*, pages 15–24.

Molina, F. and Toval, A. (2009). Integrating Usability Requirements That Can Be Evaluated in Design Time into Model Driven Engineering of Web Information Systems. *Advances in Engineering Software*, 40(12):1306–1317.

Mood, A. M., Graybill, F. A., and Boes, D. C. (1974). *Introduction to the Theory of Statistics*. McGraw-Hill, 3 edition.

Mund, J., Fernández, D. M., Femmer, H., and Eckhardt, J. (2015). Does Quality of Requirements Specifications Matter? Combined Results of Two Empirical Studies. In *Proceedings of the 9th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10.

Mylopoulos, J., Chung, L., and Nixon, B. (1992). Representing and Using Non-Functional Requirements: A Process-Oriented Approach. *IEEE Transactions on Software Engineering*, 18(6):483–497.

Ncube, C. (2000). *A Requirements Engineering Method for COTS-based Systems Development*. PhD thesis, City University.

Neubeck, P. (2012). *A Probabilistic Theory of Interactive Systems*. PhD thesis, Technische Universität München.

Nuseibeh, B. (2001). Weaving Together Requirements and Architectures. *Computer*, 34(3):115–117.

Nuseibeh, B. and Easterbrook, S. (2000). Requirements Engineering: A Roadmap. In *Proceedings of the ICSE Conference on The Future of Software Engineering*, pages 35–46.

Ott, D. (2012). Defects in natural language requirement specifications at Mercedes-Benz: An investigation using a combination of legacy data and expert opinion. In *Proceedings of the 22nd International Requirements Engineering Conference (RE)*, pages 291–196.

Paech, B. and Kerkow, D. (2004). Non-functional requirements engineering-quality is essential. In *Proceedings of the 10th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*, pages 237–250.

Palomares, C., Quer, C., Franch, X., Guerlain, C., and Renault, S. (2012). A Catalogue of Non-Technical Requirement Patterns. In *Proceedings of the 2nd International Workshop on Requirements Patterns (RePa)*, pages 1–6.

Parnas, D. L. (1972). On the Criteria to Be Used in Decomposing Systems into Modules. *Communication of the ACM*, 15(12):1053–1058.

Peterson, L. L. and Davie, B. S. (2011). *Computer Networks, Fifth Edition: A Systems Approach*. Morgan Kaufmann Publishers Inc., 5 edition.

Pohl, K. (2010). *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer.

Pollard, D. (2002). *A Users's Guide to Measure Theoretic Probability*. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press.

Rajan, S., Shankar, N., and Srivas, M. (1995). An Integration of Model-Checking with Automated Proof Checking. In *Proceedings of the 7th International Conference on Computer-Aided Verification (CAV)*, pages 84–97. Springer Berlin Heidelberg.

Rappaport, T. S. (2002). *Wireless Communications, Principles and Practice*. Prentice Hall, 2 edition.

Renault, S., Méndez Bonilla, Ó., Franch Gutiérrez, J., and Quer Bosor, M. C. (2009). A Pattern-based Method for building Requirements Documents in Call-for-tender Processes. *International journal of computer science & applications*, 6(5):175–202.

Robertson, S. and Robertson, J. (1995). Volere Requirements Specification Templates. Available online at `http://www.volere.co.uk`.

Robertson, S. and Robertson, J. (2012). *Mastering the requirements process: Getting requirements right*. Addison-wesley.

Rodrigues, G. N., Rosenblum, D. S., and Uchitel, S. (2005). Reliability Prediction in Model-Driven Development. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, pages 339–354.

Rojas, R. (1997). Konrad Zuse's legacy: the architecture of the Z1 and Z3. *IEEE Annals of the History of Computing*, 19(2):5–16.

Sommerville, I. (2007). *Software Engineering: 8th Edition*. Pearson Education Limited.

Sommerville, I. and Kotonya, G. (1998). *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, Inc.

Sommerville, I. and Sawyer, P. (1997). *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, Inc.

Stalhane, T. and Wien, T. (2014). The DODT tool applied to sub-sea software. In *Proceedings of the 22nd International Requirements Engineering Conference (RE)*, pages 420–427.

Stol, K., Raph, P., and Fitzgerald, B. (2016). Grounded Theory in Software Engineering Research: A Critical Review and Guidelines. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 120–131.

Supakkul, S., Hill, T., Chung, L., Tun, T. T., and do Prado Leite, J. C. S. (2010). An NFR Pattern Approach to Dealing with NFRs. In *18th International Requirements Engineering Conference (RE)*, pages 179–188.

Svensson, R. B., Gorschek, T., and Regnell, B. (2009). Quality requirements in practice: An interview study in requirements engineering for embedded systems. In *Proceedings of the 15th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*, pages 218–232.

Van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In *Proceedings of the 5th International Symposium on Requirements Engineering (RE)*, pages 249–262.

Vogelsang, A. (2015). *Model-based Requirements Engineering for Multifunctional Systems*. PhD thesis, Technische Universität München.

Vogelsang, A., Femmer, H., and Winkler, C. (2015). Systematic Elicitation of Mode Models for Multifunctional Systems. In *Proceedings of the 23rd International Requirements Engineering Conference (RE)*, pages 305–314.

Vogelsang, A., Femmer, H., and Winkler, C. (2016). Take Care of Your Modes! An Investigation of Defects in Automotive Requirements. In *Proceedings of the 22nd International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*, pages 161–167.

Wada, H., Suzuki, J., and Oba, K. (2010). A Model-Driven Development Framework for Non-Functional Aspects in Service Oriented Architecture. *Web Services Research for Emerging Applications: Discoveries and Trends*, pages 358–389.

Wagner, S., Lochmann, K., Heinemann, L., Kläs, M., Trendowicz, A., Plösch, R., Seidl, A., Goeb, A., and Streit, J. (2012). The Quamoco Product Quality Modelling and Assessment Approach. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 1133–1142.

Wandeler, E., Thiele, L., Verhoef, M., and Lieverse, P. (2006). System architecture evaluation using modular performance analysis: a case study. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):649–667.

Wiegers, K. and Beatty, J. (2013). *Software Requirements*. Pearson Education.

Withall, S. (2007). *Software Requirement Patterns*. Microsoft Press.

Wohlrab, R., de Gooijer, T., Koziolek, A., and Becker, S. (2014). Experience of pragmatically combining RE methods for performance requirements in industry. In *Proceedings of the 22nd International Requirements Engineering Conference (RE)*, pages 344–353.

Yang, Z., Li, Z., Jin, Z., and Chen, Y. (2014). A Systematic Literature Review of Requirements Modeling and Analysis for Self-adaptive Systems. In *Proceedings of the 20th International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ)*, pages 55–71.

Young, R. R. (2004). *The Requirements Engineering Handbook*. Artech House.

Zave, P. (1997). Classification of Research Efforts in Requirements Engineering. *ACM Computing Surveys (CSUR)*, 29(4):315–321.

Zhu, L. and Gorton, I. (2007). UML Profiles for Design Decisions and Non-Functional Requirements. In *Proceedings of the 2nd Workshop on Sharing and Reusing Architectural Knowledge Architecture, Rationale, and Design intent (SHARK-ADI)*, pages 8–15.

Zhu, L. and Liu, Y. (2009). Model Driven Development with Non-Functional Aspects. In *Proceedings of the Workshop on Aspect-Oriented Requirements Engineering and Architecture Design (EA)*, pages 49–54.

Appendix

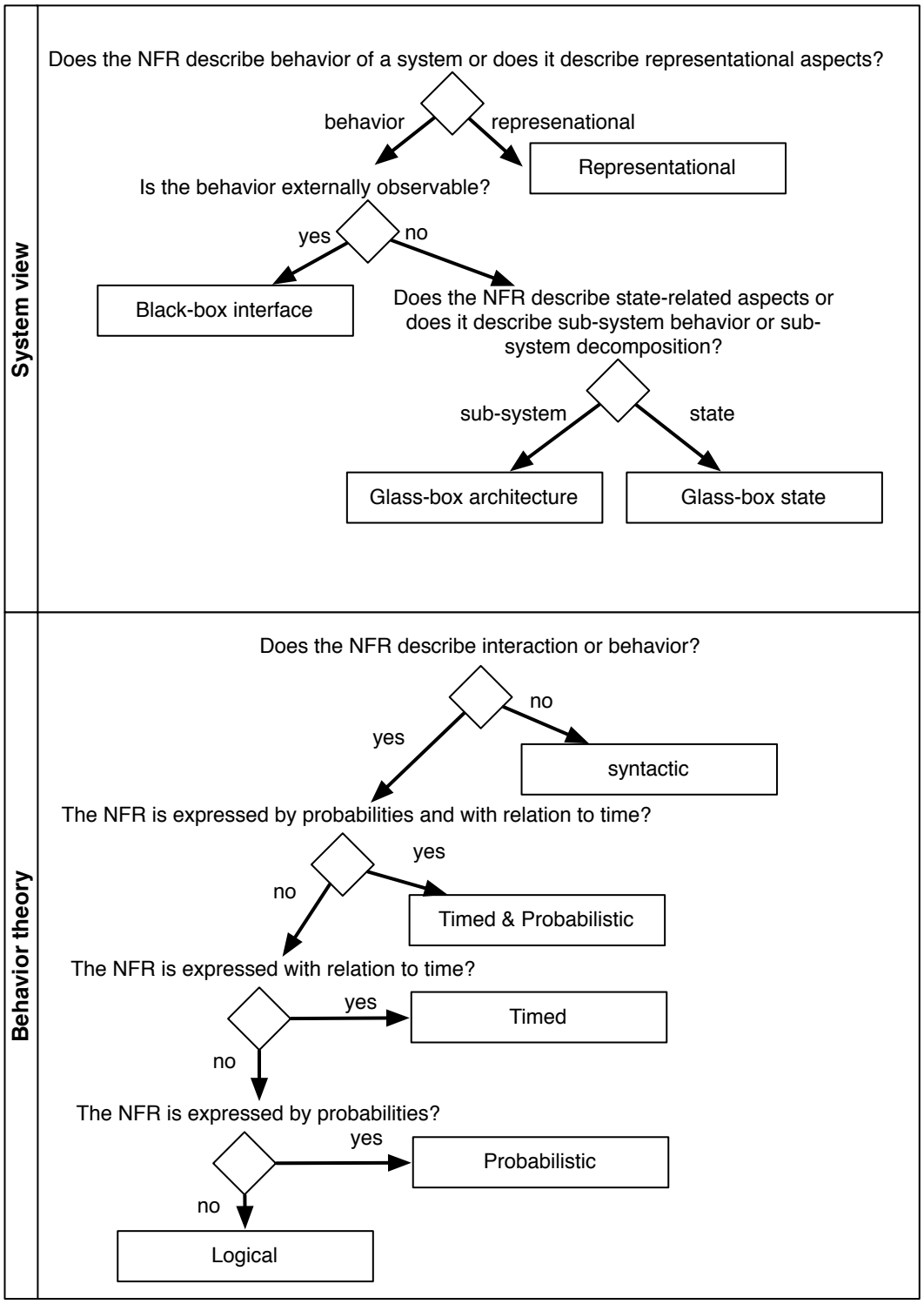


Figure A.1.: The decision tree used for the study in Chapter 5

PatternID	Pattern	ISO Quality Characteristic	System View	Behavior theory	Rationale
1	"response time shall be minimized"	Efficiency - Time behavior		Timed	A notion of time is necessary to assess the property
2	"The system shall run/be installed on platform X"	Portability - Installability	Representational	-	Describes requirements how the system (code) must be represented to be executable on a specific platform.
3	"The system shall be adaptable to environment A and environment B"	Portability - Adaptability			In contrast to the pattern above, the focus here lies on the capability of the system to be adapted for differnt environments.
4	The requirement references a standard		Representational	-	No specific behavior is described
5	The requirement references a standard and it is not clear what the standard is about	Functionality - Functional Compliance			This is the most general class of compliance classes.
6	The requirements describes a profile of expected system usage (e.g., number of users, interactions per hour)		Glass-box, Architectur	Logical	The requirement states conditions under which the system must be able to perform the desired level of performance. That means, it is a logical expression about the capability of the system design (thus, glass-box and architecture)
7	A requirement about specific data that must be present when the system is used.		Glass-box, State	Syntactic	The state space of a system includes data that is stored in a system. The requirement talks about specific (data) states that must exist. It does not speak about manipulation of the data (i.e., glass-box and state)
8	"The system must provide function X"		Black-box interface	Syntactic	This requirement is an assertion about the externally visible behavior, however, it only demands the existance of a functional entity within the functional architecture of the system. The requirment says nothing about actual interaction behavior.
9	"The system must be able to perform X"		Black-box interface	Logical	This requirement is an assertion about the externally visible behavior. It's a logical statement about desired interactions visible at the interface of the system.
10	A requirement about desired hours of operation (Availability)	Reliability - Maturity			The standard names availabilty as a combination of maturity, fault tolerance and recoverability
11	"The system interface must support the following languages: English, German, ..."	Usability - Operability	Representational	-	Language support enables the user to control the system. The requirement makes a statement about the representation of system input and outputs.
12	A requirement about how and between what data of the system shall be processed/transmitted/exchanged		Glass-box, Architecture		The requirement is not about the data but about communication between entities.
13	A requirement about properties of data (e.g., a role must be assigned to each user)		Glass-box, State		The state space of a system includes data that is stored in a system. The requirement talks about the character of data and thus about the character of the state space.

Figure A.2.: The pattern list used for the study in Chapter 5