Defining and Assessing Software Quality by Quality Models

Klaus Lochmann

Technische Universität München

Institut für Informatik
der Technischen Universität München

Defining and Assessing Software Quality by Quality Models

Klaus Lochmann

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:	Univ.-Prof. Bernd Brügge, Ph. D.
Prüfer der Dissertation:	
1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2.	o. Univ.-Prof. Dr. Gustav Pomberger Johannes Kepler Universität Linz / Österreich

Die Dissertation wurde am 31.07.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 11.12.2013 angenommen.

Abstract

Software quality plays an important role for companies today, as it is in a direct relation to the costs arising in the lifecycle of software. However, the notion of quality is diverse. A software maintenance organization, for instance, defines high quality of software as enabling effective maintenance of it. For operators of computing centers, high quality means that the available computing and memory resources are efficiently used. An end user experiences software as high quality when it supports his tasks in an effective and efficient manner and thus reduces effort for him.

In software engineering research, a lot of work has already been dedicated to the topic of software quality and quality assurance. However, we claim that there is no generally accepted definition of quality. The quality models providing taxonomical definitions face several problems, such as being ambiguous, overlapping, and incomplete. This prevents the quality models from supporting the definition of useful quality requirements. Moreover, these quality models remain on a high level of abstraction and hence are not suitable for conducting quality assessments. As a consequence of these shortcomings, today, quality assessment techniques are applied independently from the definition of quality by quality models. This results in barely comprehensible assessment results, because overall quality statements are either not provided, are missing explanation and rationale, and are not grounded in previously defined quality requirements.

This thesis presents a quality modeling approach for defining quality in a precise and assessable way. We propose an explicit quality meta-model describing the structure of quality models. It defines a product model of software systems, which provides a well-structured backbone for defining a clear and unambiguous hierarchy of quality-influencing properties. The impact of these properties on general quality attributes is explicitly captured and justified. For establishing general quality attributes of software, we rely on the activity-based paradigm, which describes quality as the capability of software to support activities conducted with it. The usage of activities has the advantage that there are clear decomposition criteria for them, resulting in a clear structure of quality attributes.

We further provide an approach for quality assessments based on the quality meta-model. It uses existing measures and analysis tools for quantifying the satisfaction of properties defined in a given quality model. For this purpose, the approach defines utility functions for evaluating the measurement values regarding the satisfaction of a property and aggregation specifications for getting an overall quality statement. The aggregation is carried out alongside the hierarchical structure of the properties and quality attributes. This way, the aggregation follows the well-defined and justified hierarchies of the quality model. Furthermore, the quality assessment approach accounts for several challenges experienced in practice, such as incomplete measurement data.

We build tool support for creating and editing quality models as well as for conducting automated quality assessments. Using this tool support we evaluate our approach: We build a quality model for Java source code and apply it to a large number of open source software systems.

Acknowledgements

First of all, I would like to express my gratitude to Prof. Dr. Dr. h.c. Manfred Broy for giving me the opportunity to work at his chair for the past five-and- a-half-years. I enjoyed the great atmosphere at this chair, the freedom he granted us, and the opportunity to work together with industry. Moreover, I would like to thank him for his supervision of my thesis and for his critical comments that made this work much better. I would also like to thank Prof. Dr. Pomberger, who agreed to act as second supervisor, for his efforts in reviewing my thesis.

In addition, my thanks go to all the colleagues with whom I had the pleasure of working. Special thanks go to Sebastian Winter, who, having already supervised my diploma thesis, introduced me to the chair and gave me valuable ideas for my thesis in countless discussions in the Quamoco project. My thanks go also to the other team-members working on Quamoco at our chair: Stefan Wagner, Florian Deißenböck, Markus Herrmannsdörfer, Elmar Jürgens, and Lars Heinemann; and to all participants of Quamoco from other organizations, especially to Andreas Göb, Alois Mayr, Christian Körner, and Michael Kläs, with whom I worked most closely. Without all the discussions and all the joint work in that research project, my dissertation would not have been possible.

At the chair of Prof. Broy, I have worked together with many people on several research and publication projects. I would like to thank them all for the good working atmosphere and the stimulating environment they provided. My special thanks go to Stefan Wagner and Daniel Méndez Fernández, who introduced me to scientific working and publishing. Moreover, I would like to thank all the other co-authors with whom I wrote and published papers.

Furthermore, I would like to thank all colleagues I was working with in the last stages of my thesis, especially those who supported me morally during lunchtime and by reviewing and giving feedback to my thesis: Alarico Campetelli, Mario Gleirscher, Maximilian Irlbeck, and Fiona Zhu.

Last but not least, I am grateful to my family – my parents and my sister – who always support me. Without them neither my study of computer science, nor the dissertation, would have been possible.

Contents

1 Introduction

The quality of software plays an important economic role today [19]. It is critical for the competitiveness and survival of companies [46, p. 4]. This is because quality is strongly connected to the costs arising during development, maintenance, and use of software. Research in software engineering has shown that preventing defects early in the development is less expensive than correcting them later [46, p. 5]. Maintenance of software amounts for about 70% of total lifecycle costs [49, p. 6]. Thus, the quality of being maintainable has a major influence on the costs incurred by software. Finally, the quality of the software has a major influence on the costs of using the software. During usage, software failures may occur, causing failures of physical systems, financial losses, or even loss of human life [121, p. 4]. Besides failures, inefficient software may also cause high costs due to unnecessarily high computing times or high memory usage. Indirect costs incurred by low software quality may be attributed to unsatisfied customers, damage to the companies' reputation, and loss of market opportunities.

Definition of Quality Notwithstanding the prime importance of software quality, there is no generally accepted definition of quality. Often different views on quality are taken, each with a different line of argument [43]. Common standards such as IEEE 610 [61], ISO 9000:2005 [68], and ISO 25010 [65] define quality as the degree to which a product satisfies *requirements*, *implied needs*, *customer and user needs*, and/or *expectations*. In this thesis, we understand quality as the satisfaction of requirements, whereby we interpret requirements in a broad sense. They do not have to be explicitly documented and they may originate from diverse stakeholders. The interpretation of quality as conformity to requirements leads to a sharp definition, which nonetheless underlines the relativeness of quality. It depends on the stakeholders of the product and their requirements, both of which may change over time. Based on this insight, defining software quality in a concrete way means capturing common stakeholders and their usual requirements. For instance, in the case of business information systems a common stakeholder for a software system is the end user, who expects the software system to support his business processes best. Another stakeholder is the operator, who is concerned with installation and configuration of the software and with the operation of the computing center. He expects the software to support his tasks during operation and to efficiently use the resources of his computing center. A third common stakeholder is the maintainer, who is concerned with modifying the system. For types of software other than business information systems, there is a similar variety of stakeholders. For each of these stakeholders, high quality means that the software satisfies the stakeholders' specific requirements.

Quality Assurance In the area of software engineering, a lot of work has already been dedicated to the topic of software quality and quality assurance. Software quality assurance is defined as "all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements" [61]. According to IEEE 12207 [64], quality assurance is concerned with both the software product and the development process. In this thesis, we focus exclusively on product quality and not on process quality. The tasks of software quality assurance are usually divided into constructive and analytical tasks [164]. Constructive quality assurance has the task of applying technical and organizational means during development to ensure that the requirements are satisfied. Analytical quality assurance has the task of evaluating and verifying the conformity of the software product to the requirements.

There are two main challenges regarding quality definition and assessment: First, to specify the requirements in a clear and unambiguous way and second, to ensure the requirements are precise enough to be measurable, i.e., a quality assessment of a product must be conductible based on the requirements. These two challenges are also reflected in the title of this thesis: *Defining quality* refers to the challenge of specifying the requirements, while *assessing quality* refers to the challenge of measuring the satisfaction of the requirements.

1.1 Problem Statement

In literature, quality models have been proposed for specifying the requirements regarding software quality. The first quality models, dating back to the 1970's, follow a taxonomical approach: They use a decomposition of the concept *quality* into quality attributes such as *maintainability* and *reliability*. Based on these hierarchical quality models, the international standards ISO 9126 [70] and ISO 25010 [65] have been developed. The quality attributes defined by such quality models are often criticized for being ambiguous, overlapping, and incomplete [1, 26]. Later work selectively addresses these shortcomings. For instance, Deissenboeck et al. [33] introduce a clear principle of decomposition for the quality attribute *maintainability*, relying on activities. In general, current quality models lack a clear definition of all relevant quality attributes. Furthermore, these quality models define quality attributes at a high level of abstraction. Some approaches [39, 129, 133, 155] attach measures directly to the quality attributes, in order to operationalize them. Due to the large difference in the level of abstraction between the quality attributes and the measures, these approaches face severe problems in practice. Thus, to date there is no feasible approach to conducting quality assessments with such quality models.

Irrespective of the research done on quality models, the discipline of software measurement emerged [36]. It takes more of a bottom-up approach by defining measures for characteristics of software. These measures mostly refer to source code and tools have been implemented to automatically calculate and visualize them. However, due to the bottom-up approach, it is usually easily determinable characteristics that are measured, instead of relevant characteristics [26, p. 50]. Thus, the relation of the measures to software quality is unclear, which hinders the interpretation of the measurement values. Based on the measures, dashboard tools for visualizing the measurement values have emerged (e.g. [28, 149]). These tools present the single measurement values and some aggregations, while leaving the interpretation of the data to the user [158, 159]. Thus, there is no systematic way of getting to an overall quality assessment of a software product.

Summing up, the definition of quality and the measurement of quality are rather separate topics today. Quality models define high-level requirements in the form of quality attributes, which are not measurable. On the other hand, a large number of measures and tools for implementing them are available without the link to the quality models.

1.2 Contribution

In this thesis, we propose a quality modeling and assessment approach addressing the shortcomings of existing approaches in the following ways.

Quality Meta-Model

Quality models have the task of defining quality-properties of software products on different levels of abstraction and to clarify the interrelations between these quality-properties. For developing clear quality models, an explicit definition of the concepts they consist of is necessary. To this end, we introduce a quality meta-model describing the structure and meaning of quality models. The quality meta-model is defined in a formal way to assure an unambiguous definition of its concepts.

An important concept in our quality models is to use a product model for structuring purposes. The product model defines the artifacts of which a software product consists. The definition of quality-properties in the quality model relies on the artifacts of the product model. Moreover, the decomposition of abstract quality-properties into more concrete ones is aligned with the product model. This way, a structuring mechanism is offered, which gives guidance for decomposing abstract quality-properties and offers a way of organizing the large number of quality-properties defined in realistic quality models.

Quality Assessment Approach

Based on the quality meta-model, we introduce a quality assessment approach. It defines how measures and the tools for implementing them are connected to the quality model and how measurement results are aggregated for an overall quality statement. For this purpose, the assessment approach defines how measurement results are evaluated and how specifications for aggregating evaluation results are defined. By integrating measures into the quality model, the relation of the measures to the quality-properties is made explicit and justified by a prose description.

The quality assessment approach accounts for three challenges experienced in practice: First, it provides an approach for defining parameters for utility functions, i.e., threshold values for measurement values. Second, it defines how incomplete measurement data can be handled. Third, it addresses the problems encountered when integrating existing tools by enabling a bottom-up quality assessment and by supporting the use of rule-based static code analysis tools.

Quality Model Instances

First, based on the quality meta-model, an instance of a quality model is constructed. It provides a general definition of quality attributes being applicable to a wide range of software products. These quality attributes are structured according to the activity-based paradigm known from literature. Thereby, overlappings and ambiguities in the definition of the quality attributes by ISO 25010 are identified and resolved.

Second, the quality model defining the quality attributes is expanded to a quality model for Java source code. It defines properties of source code and measures for quantifying them by existing tools. Furthermore, it contains all needed aggregation specifications and parameters of the utility functions for automated quality assessments.

Tool Support

We provide a *quality model editor* for creating and editing quality models conforming to the meta-model. It provides a graphical user interface supporting different views on quality models and it automatically checks constraints to support the quality modeler. For conducting quality assessments based on the quality model a *quality assessment tool* is provided. It executes external analysis tools for obtaining measurement data and aggregates the data according to the specifications in a given quality model.

Evaluation

Using the quality model for Java source code, we demonstrate the applicability of our quality modeling and assessment approach:

- We show that the meta-model and the corresponding tools enable the creation of realistic quality models with several hundred model elements. The resulting quality model is well manageable with our tool support.
- The automatic quality assessment of software systems leads to a satisfactory differentiation of systems. This means software systems of different quality levels generate different quality assessment results.
- The quality assessment results using the quality model are in concordance with an expert assessment for five systems. Thus, we conclude that our approach is able to produce valid quality assessment results.
- We show by example that the quality assessment based on the quality model enables tracing of quality changes between different versions of a software system. This allows developers to identify the origin of a change in the quality assessment.

1.3 Outline

Chapter 2 introduces the terminology used in this thesis. It focuses on the definition of the term *quality* and related terms. Furthermore, the role of quality models in quality assurance tasks is discussed in detail. In Chapter 3, we summarize and discuss the state-of-the-art regarding quality models and software measurement and highlight the shortcomings of current approaches. Furthermore, we report on a survey conducted in industry to assess the current state of practice regarding the aforementioned topics. Chapter 4 introduces the formal definition of the quality meta-model. Based on that, Chapter 5 presents the quality assessment approach. In Chapter 6 we present both the general quality model describing the quality attributes and the quality model for Java source code. Chapter 7 describes the tool support we developed. It introduces both the quality model editor and the quality assessment tool. Chapter 8 reports on the case study conducted with the quality model for Java source code. Chapter 9 summarizes the main contributions of this thesis and Chapter 10 shows further research directions.

Previously Published Material

The material covered in this thesis is based, in part, on our contributions in [27, 32, 47, 88, 89, 99–104, 158–163].

2 Preliminaries

The quality of software plays an outstanding economic role today [19]. It is critical for the competitiveness and survival of companies [46, p. 4]. The consequences of software quality problems are exemplified by spectacular failures of software with dire consequences [121, p. 4], for instance the failure of the Ariane 5 rocket. Beside failures, software quality is also related to the costs arising in the lifecycle of software. It is generally known that preventing defects early in the development is less expensive than correcting them later [46, p. 5]. The growing economic importance of software quality can also be seen by the constant increase of efforts spent for software maintenance [26, p. 15], reaching about 70% of total lifecycle costs [49, p. 6].

Despite the basic insight from literature that software quality is related to failures of the software but also to the lifecycle costs of it, a universal definition of quality is still missing. In the first section of this chapter, we introduce a comprehensive discussion of the term quality and of related terms. In the second section, we will discuss the topic of software quality assurance and discuss our focus on product quality. We also explain the need for quality models and measurement procedures. A detailed coverage of the state-of-the-art can be found in Chapter 3.

2.1 Terminology

In literature, different definitions of *software quality* are proposed. Still, no generally accepted definition has emerged. Garvin [43] discusses different views on quality and concludes that quality is a "complex and multifaceted concept". In this section, we discuss the definition of *quality* and related terms as they are used in this thesis.

2.1.1 Properties & Quality

A basic concept for talking about quality is that of a *property*. In philosophy, there is consensus that "properties include the attributes or qualities or features or characteristics of things" [145]. While *property* is often used as the root term, *attribute*, *quality*, *feature*, and *characteristic* are used as synonyms for it, depending on the context.

A typical example for a property is *redness* of an *apple*. We say that the *apple* exemplifies the property *redness*. The main difference between things and properties "is that properties can be [. . .] *exemplified*", whereas things cannot [145]. Thus, a property is always referring to an object.

Besides the use of the term *quality* as a synonym for *property*, in its common usage it is employed in a judgmental way, referring to the property of being useful for a certain purpose (*fitness for purpose*) [57]. In business, the purpose is usually to satisfy a customer, leading to the statement that "quality means those features of products which meet customer needs and thereby provide customer satisfaction" [79].

According to this definition, quality is a relative concept, which depends on the circumstances in which it is invoked [57]. The purpose of a system depends on a stakeholder who pursues the purpose, whereby the purpose is different for each stakeholder. For instance, for the end user, the system has the purpose of supporting his tasks, while for a maintainer the purpose is maintenance.

This definition remains in opposition to defining quality as an absolute "innate excellence", as in the transcendent view on quality found in Garvin [43]. It mostly coincides with the user-based view of Garvin, which says quality "lies in the eyes of the beholder". Thus, it equates quality with the satisfaction of the stakeholder.

Since our definition of quality relies on the purpose of the system for certain stakeholders, describing that purpose is essential. Because software systems usually provide a benefit by interacting with their environment, describing usage processes – i.e., tasks supported by the software – is a suitable form for describing a large number of purposes. This is reflected by the common use of *use cases* for describing how a business information system supports the tasks of its end users [13]. However, it can also be applied to maintainers, who conduct maintenance tasks with the system and see the purpose of the system as enabling effective maintenance [33].

However, there are also qualities going beyond what are describable by tasks. For instance, the satisfaction of a stakeholder by the product may go beyond the support of tasks: Being attractive to an end user is a typical example of such a property.

Since properties can be defined regarding different types of objects, we distinguish between three types of properties. First, a property can describe a *stakeholder*. Since stakeholders are usually people or organizations, typical properties for characterizing stakeholders are *satisfaction*, *trust*, and *pleasure*. For instance, with *satisfaction of the end user* we describe whether the end user of the software system is satisfied with the experience of using it. Second, a property can be used to describe a process of the life-cycle of software, which especially includes the development and usage of the software. Such properties are, for instance, *efficiency* of conducting a certain use case with support of the software, or *effectiveness* of conducting maintenance tasks on the software. Third, a property can describe the software product or parts of it. Typical properties describing the product are the quality attributes of ISO 25010, like *usability*, *maintainability*, or *reliability*. Properties referring to parts of the product may also be of a more technical nature, describing, for instance, the adequateness of the size of fonts in a graphical user interface, or the nesting depth of source code.

We in general assume that these types of properties influence each other as depicted by Figure 2.1. The software product is used in different processes to fulfill its tasks. Thus, the product properties influence the process properties. The process is conducted by a stakeholder to fulfill a certain purpose. Thus, the process properties influence the stakeholder properties. Additionally, the product properties can influence the stakeholder properties directly.

An example for such a chain of properties could be: The *usability* of the software (product property) influences the *efficiency* of a specific usage process (process property), which influences the *satisfaction* of the end user (stakeholder property). A direct influence of a product property on a

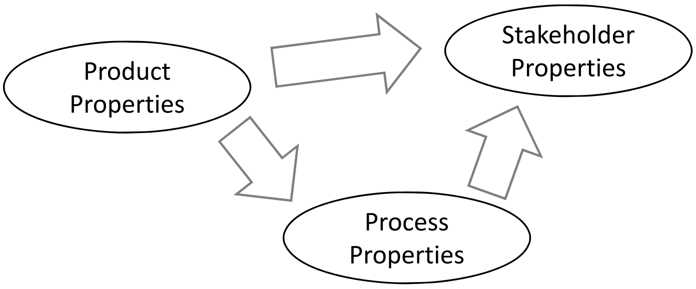


Figure 2.1: Influences between the Types of Properties

stakeholder property could be: The attractiveness of the software product directly influences the satisfaction of the user.

It is important to note that in practice it is usually not possible to specify the influencing properties in this chain to such an extent that they can be used to fully express the influenced properties.

2.1.2 Requirements & Quality

In our discussion of the notion of properties and quality in the previous section, we came to the understanding that quality of a product is the property of being useful for a certain purpose. Being useful for a purpose means that the product possesses properties the stakeholders require for pursuing their purposes.

A property that is required by a stakeholder is called a *requirement*. Since a requirement refers to a property, the three types of properties can be transferred to different types of requirements: A *stakeholder requirement* refers to a stakeholder property; for instance, “The end user must be satisfied with the software”. A *process requirement* refers to a process property; for instance, “The execution of the use case ‘book flight’ must be executed in less than 10 minutes on average”. A *product requirement* refers to a product property; for instance, “The font size in the graphical user interface must be at least 12 points”.

Analogously to the chain of influence between product, process and stakeholder properties, the requirements of these three types are based on each other. In general, in software engineering, the primary stakeholder is the business developing the software product. This business usually has the requirement of achieving satisfaction of the customers (stakeholder requirement). Since the customer uses the software for supporting his tasks and customer satisfaction is required, the new requirement of efficiently supporting the users’ tasks is derived. This clearly is a process requirement. From such process properties, product properties are derived by stating, for instance, the functional requirements of the software system.

The formulation of requirements as such a chain is also common in requirements engineering approaches. For instance, the approach of Mendez et al. [110] starts with a business specification, which states the mission and objectives of the business. Based thereon, the processes conducted in the business are defined. The requirements in this business specification correspond to the stakeholder and process requirements in our terminology. Based on the business specification, the approach of Mendez et al. defines the scope of the software system, the use cases of the software, and the detailed technical requirements; all these are part of the software requirements specification. This specification consists of product requirements, in our terminology.

Our definition of *requirement* is very general and does not define whether requirements are implicitly present at a stakeholder, whether they are explicitly documented, or whether they are documented in a system specification accepted by all stakeholders. Thus, our definition of requirement is in line with the IEEE Glossary [61], defining a requirement as "(1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A 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) A documented representation of a condition or capability as in (1) or (2)". Furthermore, the definition of ISO-9000:2005 [68] is in line with our definition, defining a requirement as a "need or expectation that is stated, generally implied or obligatory".

The requirements of a stakeholder are present, regardless of whether they are documented or not. However, for developing software it is necessary to document them explicitly, which is done in the *requirements elicitation*. Requirements elicitation is the process of discovering the requirements of a stakeholder and explicitly documenting them. This definition conforms to usual definitions found in the requirements engineering literature [90, 131, 139]. However, the literature usually considers the identification of all stakeholders as a part of elicitation, while we define the elicitation as a process applied individually to each previously defined stakeholder. Since each stakeholder has his own purposes regarding the system, the elicitation takes place individually for each stakeholder and the requirements are individual to each stakeholder.

In system development, however, all stakeholders have to agree on a common set of requirements. The process of deciding which requirements of individual stakeholders are accepted for the system development is called *requirements negotiation*. The negotiation also includes the assignation of priorities to the requirements according to their importance for different stakeholders. The result of the requirements negotiation is a *requirements document*, which is an artifact containing all the requirements accepted for the system development.

According to the above definitions, we distinguish between three categories of requirements, as summarized in Figure 2.2. First, the *implicit requirements* are all the requirements a stakeholder has, even if not explicitly stated. Second, the *explicit requirements* are the explicitly documented requirements of a stakeholder. The explicit requirements result from the requirements elicitation. The requirements elicitation may not necessarily be able to explicitly document all implicit requirements. Third, the *accepted requirements*, which are documented in the requirements specification, are the requirements of all stakeholders, resulting from the *requirements negotiation*. Due to necessary compromises and trade-offs, not all requirements of all stakeholders may be part of the accepted requirements. Nonetheless, the accepted requirements are the basis for the software development.

Note that each of these categories of requirements can refer to the three types of properties (stakeholder, process, product). An implicit requirement, for instance, can refer to a stakeholder, process, or product property.

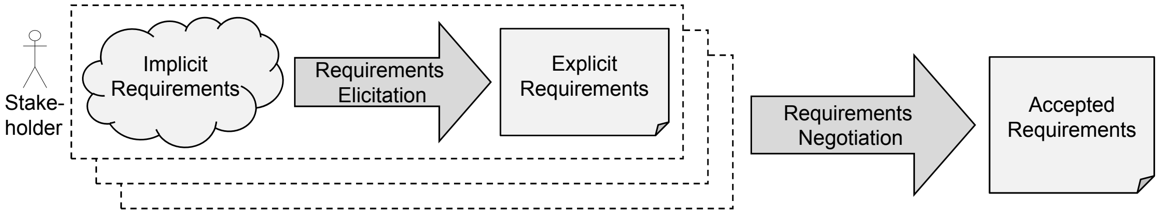


Figure 2.2: Overview of Categories of Requirements

2.1.3 Different Views on Quality

Our considerations so far lead to the deduction that *quality means satisfaction of the requirements of a stakeholder*. This is because we understand quality as being fit for purpose and being fit for purpose means satisfying requirements. This deduction is in line with our interpretation of quality as a relative concept. The requirements are dependent on a certain stakeholder, just like the fitness for purpose depends on the stakeholder pursuing the purpose.

As we have seen in the previous section, there are different categories of requirements. According to them, we define two different views on quality:

1. stakeholder's view on quality, based on the implicit requirements
2. producer's view on quality, based on the accepted requirements

The *stakeholder's view on quality* takes the implicit requirements of a certain stakeholder into consideration. It represents the view on quality of that stakeholder, regardless of whether his requirements have been explicitly documented or not. It resembles the "user-based approach" of Garvin [43] to define quality: "[T]hose goods that best satisfy their [the users'] preferences are those that they regard as having the highest quality".

The *producer's view on quality* takes the accepted requirements into consideration. Thus, it represents the view on quality of the producer of the software, which relies on the specification as the basis for developing the software. This view resembles the "manufacturing approach" of Garvin [43], which has been common in the manufacturing industry and gave the definition that "quality was 'conformance to specification' " [79]. This definition assumes that the specification represents the actual customer needs, so that conforming to the specification means satisfying customer needs. Accordingly, "any deviation [from the specification] implies a reduction in quality" [43].

2.1.4 Functional vs. Non-functional Requirements

In software engineering, different types of requirements are often defined. A common distinction is between *functional* and *non-functional* requirements [131, 151]. This distinction is often criticized for being inaccurate and not clearly definable [45]. Nonetheless, in practice, requirements referring to the input-output behavior of the system are often called *functional requirements*. This type of requirement is usually found in textual requirements specifications in the form of use cases or scenarios. All other types of requirements are then called *non-functional requirements*.

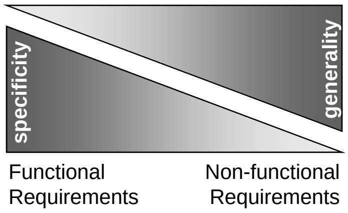


Figure 2.3: Generality/Specificity of Requirements

An important facet of the distinction between functional and non-functional requirements is that it largely corresponds to the specificity of requirements. Figure 2.3 shows the relation of the two requirement types to specificity/generality of requirements. Functional requirements are usually specific to a certain system, while non-functional requirements are applicable to a large range of systems. For instance, the functional requirement expressed by the scenario “booking a flight” is unique to a flight-booking system, while the maintainability-related non-functional requirement of having consistently named identifiers in the source code applies to virtually all software systems.

Evaluating whether requirements are satisfied is an important task in software engineering and is called *quality assessment*. A quality assessment is defined as an evaluation of whether a software system possesses certain properties; i.e., it is an evaluation of whether a software system satisfies certain requirements.

According to the discussion on different types of properties and different categories of requirements, a quality assessment can be done using different methods. For covering implicit requirements, a questioning of the stakeholder is the only viable way. Since those requirements are not documented and the stakeholder may not explicitly be aware of them, questioning the stakeholder about his satisfaction is the only feasible possibility. For both the documented requirements of one stakeholder and the accepted requirements of all stakeholders the same methods can be applied. Yet for each type of property, a different method has to be chosen: On the level of stakeholder properties, only questioning the stakeholder can directly measure the property. On the level of process properties, where qualities are defined via tasks conducted with the software, process measures can be used to assess the effectiveness of the software in supporting the tasks. On the level of product properties, the product itself is evaluated using product measures.

2.2 Software Quality Assurance

In our discussions up to this point, we have been talking about product quality, i.e., the quality of the delivered software products. However, the IEEE-12207 [64] states that the purpose of the software quality assurance process “is to provide assurance that work products and processes comply with predefined provisions and plans.” Also the IEEE-610 [61] defines software quality assurance as “(1) A planned and systematic pattern of all actions necessary to provide adequate confidence that an item or product conforms to established technical requirements. (2) A set of activities designed to evaluate the process by which products are developed or manufactured.” Both definitions mention that quality assurance must guarantee both product and process quality.

Process quality describes characteristics of the process of software development. Since the process of developing the software product has an obvious influence on the quality of the resulting product, many approaches aim at increasing the process quality. Process models such as the V-Model XT [150] and the rational unified process (RUP) [91] prescribe a detailed process for developing software. Process maturity models such as CMMI [138] and SPICE [71] provide approaches for assessing and improving the processes of organizations. The reason for focusing on the quality of processes is the assumption that better development processes will lead to better products. However, in software engineering there is little evidence for this assumption [86]. A study of the correlation of the CMMI level of a company and the number of shipped defects in its software showed that, in general, the number of defects decreases when the CMM level rose. Nonetheless, the best companies at CMM level 1 produced software with fewer defects than the worst companies at level 5 [26, p. 23] [76]. In our view, improving process quality is one technique for improving the product quality. Since the product is delivered to the customer, ultimately the product quality is of relevance. Thus, the ultimate goal is to be able to define and measure the quality of a software product independently from the process. This way, a quality seal enabling comparison of different software products would also become feasible.

As we have discussed earlier, we interpret quality as conformance to requirements. An obvious precondition for an effective quality assurance is therefore comprehensively specified requirements. The elicitation of requirements is a classical task of requirements engineering and the elicitation of quality requirements can, in particular, be seen as being part of quality assurance. Particularly, the form of documentation of the requirements is relevant for this thesis, because the documentation serves as a basis for all other quality assurance activities. The activities taken to provide confidence that the requirements are satisfied are split into two types according to Wallmüller [164]: constructive and analytical quality assurance.

The constructive quality assurance has the task of applying technical and organizational means to ensure the target quality during development [164]. According to Galin [41], a quality plan is developed that defines all the means that are applied in order to achieve quality. Thereby, the definition of the quality plan relies on the quality requirements.

The analytical quality assurance has the task of evaluating and verifying the conformance of the software product to the requirements [164]. The analytical measures are not only applied at the end of the development, but throughout the development process. Thereby, deviations from the targeted quality can be detected early and corrective measures can be taken.

2.2.1 Quality Requirements Definition

As discussed above, a precondition for conducting constructive and analytic quality assurance is to specify the quality requirements. More generally, Wallmüller [164, p.6ff] states that a prerequisite for all quality assurance activities is to define a quality plan, which comprises quality requirements for both the development process and the product. Regarding the development process, typical quality requirements may comprise the use of a certain development process, the definition of quality gates, the prescription of inspections and a release process.

The quality requirements regarding the software product are usually specified during the requirements engineering phase of a software project. A major demand regarding the quality requirements for the product is to define testing and measurement procedures in order to be able to test the satisfaction of the requirements. Being able to test the satisfaction of requirements is a prerequisite for planning and conducting the quality assurance in software development.

2.2.2 Constructive Quality Assurance

Constructive quality assurance is seen as a wide field according to Wallmüller [164, p.73ff]. It comprises all organizational the means of the software development project, such as, for instance, the chosen development process, the software configuration management, and the organizational culture. Furthermore, it comprises technical means, such as the technical infrastructure, including development tools, or testing tools.

We take a narrower look at constructive quality assurance and focus on techniques for communicating the requirements to the developers. For requirements of a more functional type (according to Section 2.1.4), classical documentation and communication means of requirements engineering are used, for instance describing input-output behavior by use cases or message sequence charts. Whereas, for a large number of requirements being more of a non-functional type (according to Section 2.1.4), guidelines are a usual means of communicating. At the technical level of programming languages, guidelines and style guides are the prime way of communicating requirements to developers [59, p. 65ff]. Guidelines describe conventions for using the programming language constructs in order to achieve different goals. The most common goal is to achieve readability and understandability of the source code. However, specialized guidelines such as MISRA-C [114] have a particular focus on enhancing reliability by forbidding certain failure-prone language constructs. The Web Content Accessibility Guideline (WCAG) [168] describes requirements for Web pages in order to make them accessible by people with disabilities. Additionally, international standards are often sources of detailed instructions on how to build a system. For instance, the ISO 26262 [72] describes safety relevant characteristics of systems for electrical passenger vehicles, the satisfaction of which is even required by law.

2.2.3 Analytic Quality Assurance

Analytic quality assurance, according to Wallmüller [164, p. 141], has the task of analyzing the quality of the software product and the development process. For assessing the development process, a large variety of process improvement and auditing techniques are available. Since this thesis focuses on product quality, we will take a detailed look at analysis techniques for product quality.

Product-based analytic quality assurance means verifying the satisfaction of the requirements by the software product. For this purpose, a large range of different techniques can be applied. Some of the techniques are only applicable to the end product, i.e., the delivered software; others, however, are also applicable to intermediary artifacts, such as design documents. Here we will give a brief overview of the different techniques and their characteristics.

A way of verifying the satisfaction of predominantly functional requirements by the software product is *dynamic testing*. Dynamic software testing means that the software is executed with specified inputs and the outputs are observed and evaluated [61]. The combination of inputs and expected outputs are called *test cases*. They are derived based on the requirements. The question of how to derive test cases from requirements and how many test cases are needed for adequate test coverage has been extensively investigated. There are, for instance, two principally different strategies for generating test cases: white-box testing and black-box testing [164, p. 173]. The white-box testing takes the inner structure of the software under consideration to define test cases, while the black-box testing does not consider the inner structure. Dynamic testing has the limitation that it can only be applied when the implementation of the tested component has been finished. On the other hand, it is possible to automate dynamic tests so that they can be repeated with little effort.

In contrast to dynamically testing the software, *statical tests* are also possible [164, p. 144]. Statical analysis can be done manually or automatically by tools. For manual tests there are different degrees of formality. Usually, an informal test is called a *review* or a *walkthrough* [46, p. 155]. A walkthrough is a rather informal technique without a fixed structure. Its objective is to find errors in development artifacts of all kinds, from requirements documents to source code. An *inspection* is a formal and more comprehensive type of manual test. It defines different roles, such as moderator, producer, reviewer, and recorder and prescribes a formally defined process. Though the effort required for conducting inspections is higher than for walkthroughs, empirical results show their cost effectiveness [44, p. 17ff] and suitability for finding faults [9]. Moreover, inspections are useful on early development artifacts [40].

In tool-based static analysis, the source code is usually analyzed. In the area of software measurement a lot of work has been dedicated to code measures [46, p. 110]. Automated analysis of code ranges from size and complexity measures, e.g., McCabe-Complexity [108], to rule-based analysis tools, e.g., FindBugs [37]. Other types of static analysis, such as clone detection [78], can also be extended to artifacts other than source code, for instance, to requirements documents [77]. As automated measurement is in focus of this thesis, a detailed review of existing work in this area can be found in Chapter 3.

2.2.4 Quality Control Cycle

The quality assurance tasks in software development are not conducted as one-time undertakings' rather they are conducted continuously. According to Deissenboeck [26, p. 87], a continuous quality control cycle is best described as a feedback loop, of the type familiar from control systems. The object under control is the software system and the output is the quality of the software product, i.e., the degree of conformance of the software product to the requirements. The control loop measures the degree of conformance to requirements (analytic quality assurance). Based on the deviations (i.e., quality deficits) corrective action is taken, which requires communicating the desired requirements (constructive quality assurance) and the deviations.

It is desirable to have as brief a cycle time as possible in the control loop. This way, quality deficits are detected and corrected early, when the effort to correct them is still low. However, the quality analysis requires effort itself and restricts how often it can be conducted.

2.2.5 The Role of Quality Models

Quality models have been developed to support the main activities of quality assurance. The main activities have been introduced in the above sections:

1. Quality Requirements Definition
2. Constructive Quality Assurance
3. Analytic Quality Assurance

Quality models in literature are often fitted to support one or more of these activities. In [32] we developed a classification scheme for quality models according to their purposes, which is similar to the categorization used here.

An important group of quality models are the quality definition models. They have the purpose of defining what constitutes software quality by defining the quality characteristics relevant for software products. The simplest of these models are given in the form of taxonomies, defining a hierarchy of quality characteristics. By specifying predefined terms, they support all three activities by giving a simple mechanism for structuring topics. However, they are especially used in requirements engineering, because the taxonomies provided by them can be used as a checklist in requirements elicitation. This way, they help to ensure that no relevant quality-related topic is omitted. More concrete quality models go beyond defining a taxonomy and provide a large collection of properties that are related to quality. Hence, they can be used as a collection of potential requirements in requirements elicitation. Such quality models have been used, for instance, for eliciting security requirements in our publication in [163].

Depending on the type of quality model, it can constitute a rough guideline for quality-related topics, but also a concrete collection of potential requirements. Accordingly, the boundary between a quality model as a sheer supporting artifact and a quality model as a central container of requirements is blurred. In the latter case, the requirements engineering has the task of tailoring an existing quality model and extending it by requirements for a specific software product. Our vision is to use a quality model as a central artifact, which specifies the quality-related requirements and provides the measurement specifications for testing the satisfaction of these requirements.

Another task of quality models is to support constructive quality assurance. This means that a quality model should be the central repository of knowledge and information about software quality. Furthermore, the quality model should be suited to communicating that knowledge and information in a way that effectively supports the project participants in software development. More specifically, this means that the quality requirements expressed by a quality model must be communicated to the developers and other participants in development in a way that they can work constructively with. For this task, specific quality models have already been developed, being present in the form of guidelines and checklists, which are targeted at developers. They often constitute a collection of best practices and rules, but lack clear rationales and relations to affected quality characteristics [33]. In our vision, such specific quality models should be integrated into one comprehensive quality model so that the definition of quality characteristics and the communication of knowledge are realized by the same quality model.

The third main task of quality models is to support analytic quality assurance. This means that a quality model defining quality characteristics or more detailed quality requirements must also define how the satisfaction of them can be tested. Typical quality definition models do not achieve this purpose, because they are too abstract. Thus, specific quality assessment models have been developed which address the topic of measurement. They mostly constitute collections of measures and tools for measurement. However, they lack a definition of the relation of the measures to quality characteristics. In our vision, the measurement specifications also have to be integrated into one comprehensive quality model. This way, for all quality characteristics and the associated quality requirements of a quality model, a concrete way of testing them would be achieved.

Using one quality model for requirements definition, communication, and analytic quality assurance would overcome shortcomings in all three areas: The quality requirements would be testable by the analytic quality assurance method; the measures of the quality assurance method would be justified by requirements; and the knowledge communicated to project participants would conform to both requirements and actually measured quality characteristics.

3 State-of-the-Art

In the first three sections of this chapter, we discuss related work regarding existing quality models, software measurement approaches, and quality requirements approaches. In the fourth section, we report on a survey we conducted to assess the state of practice regarding quality models and quality measurement. The fifth section explains the relation of this thesis to the project *Quamoco*, in which the author of this thesis was involved. In the last section, we summarize our findings and insights.

3.1 Quality Models

In literature a large number of quality models have been proposed. These quality models can be categorized according to the different purposes they pursue and according to the objects they are focusing on (e.g., product, process, resource) [32, 87]. Due to our general limitation to product quality, we focus on quality models describing the software product. We structure our discussion of quality models as in our publications [158, 159] and discuss them roughly according to their chronological appearance: First, we discuss hierarchical quality models and then we focus on richer quality models.

3.1.1 Hierarchical Quality Models

At the end of the 1970's, the first hierarchical quality models were published by McCall [109] and Boehm et al. [14]. Their objective was to define what *quality* is all about, by decomposing it into more concrete and tangible *quality characteristics*. Figure 3.1 shows the decomposition provided by Boehm et al. [14]. The *general utility* of a software product is decomposed to *portability*, *as-is utility*, and *maintainability*. These quality characteristics are further decomposed to low-level quality characteristics, such as *consistency*, *structuredness*, and *conciseness*. Boehm highlights the importance of measuring the low-level characteristics. He introduces several measures to this end, which are conductible through expert judgments. For instance, for quantifying *completeness*, some measures are:

1. Is input data checked for range errors?
2. Are loop and multiple transfer index parameters range tested before they are used?
3. Does the program allow for recovery in case of non-fatal errors?

McCall's model follows a similar approach, although with a different terminology: At the top level of the hierarchy he talks about*factors*, which are refined to *criteria*. The criteria are then quantified by *metrics*. Another similar model is FURPS [48]. FURPS is an acronym for the main quality characteristics of the model: *functionality*, *usability*, *reliability*, *performance*, and *supportability*.

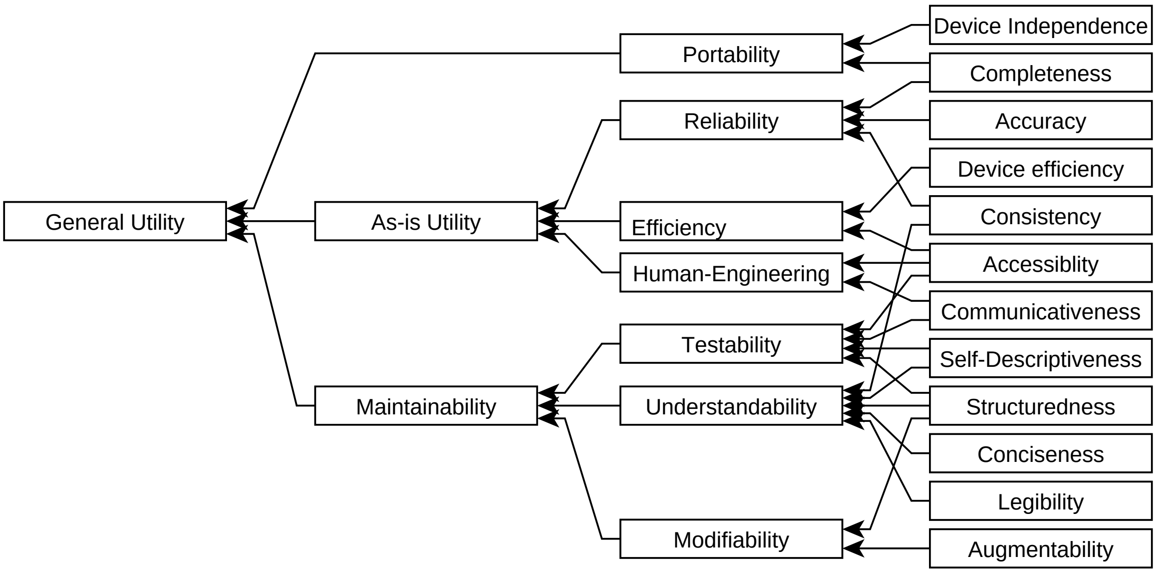


Figure 3.1: Quality Characteristics Tree of Boehm et al. [14]

Based on these works, the ISO 9126 [70] standard for software quality was published in 1991. In 2011, the successor to this standard, ISO 25010 [65] was published. The principal structure of both standards is the same. First, the ISO 25010 differentiates between *quality in use* and *product quality*. Figure 3.2 shows the quality in use model, defining “the impact that the product (system or software product) has on stakeholders” [65]. Figure 3.2 shows the product quality model. This essentially resembles the earlier hierarchical models by defining a tree of *quality attributes*. This model itself does not define a method for quantifying (or measuring) of the quality attributes. The additional standards ISO 25020 [66] and ISO 25021 [67] define a process for deriving measures for quality attributes. However, those descriptions are very general and no concrete measures are given.

After the publication of the ISO 9126, various adaptions and amendments have been proposed to it. Some publications (Franch and Carvallo [39], Van Zeist and Hendriks [155], Samoladas et al. [133]) adapt the quality attributes of ISO 9126 and add measures for quantifying them. Ortega et al. [117] proposes a different structuring mechanism for the quality attributes, based on two root characteristics, called *product effectiveness* and *product efficiency*.

Critique

Deissenboeck [26, p. 36] gives a summary of various critiques found in literature (see [1,22,58,84–86]) of the hierarchical models. Briefly, the following shortcomings of hierarchical models have been identified:

- *Ambiguity, Incompleteness, and Overlapping:* The quality attributes of these models are “ambiguously defined, incomplete with respect to quality characteristics and overlapping with respect to measured properties” [1]. A similar problem with the definition of the quality attributes is the lack of rationale for their selection and their hierarchical composition [86]. For instance, it is unclear why in the ISO 25010 *adaptability* is a sub-characteristic of *portability*, which is a

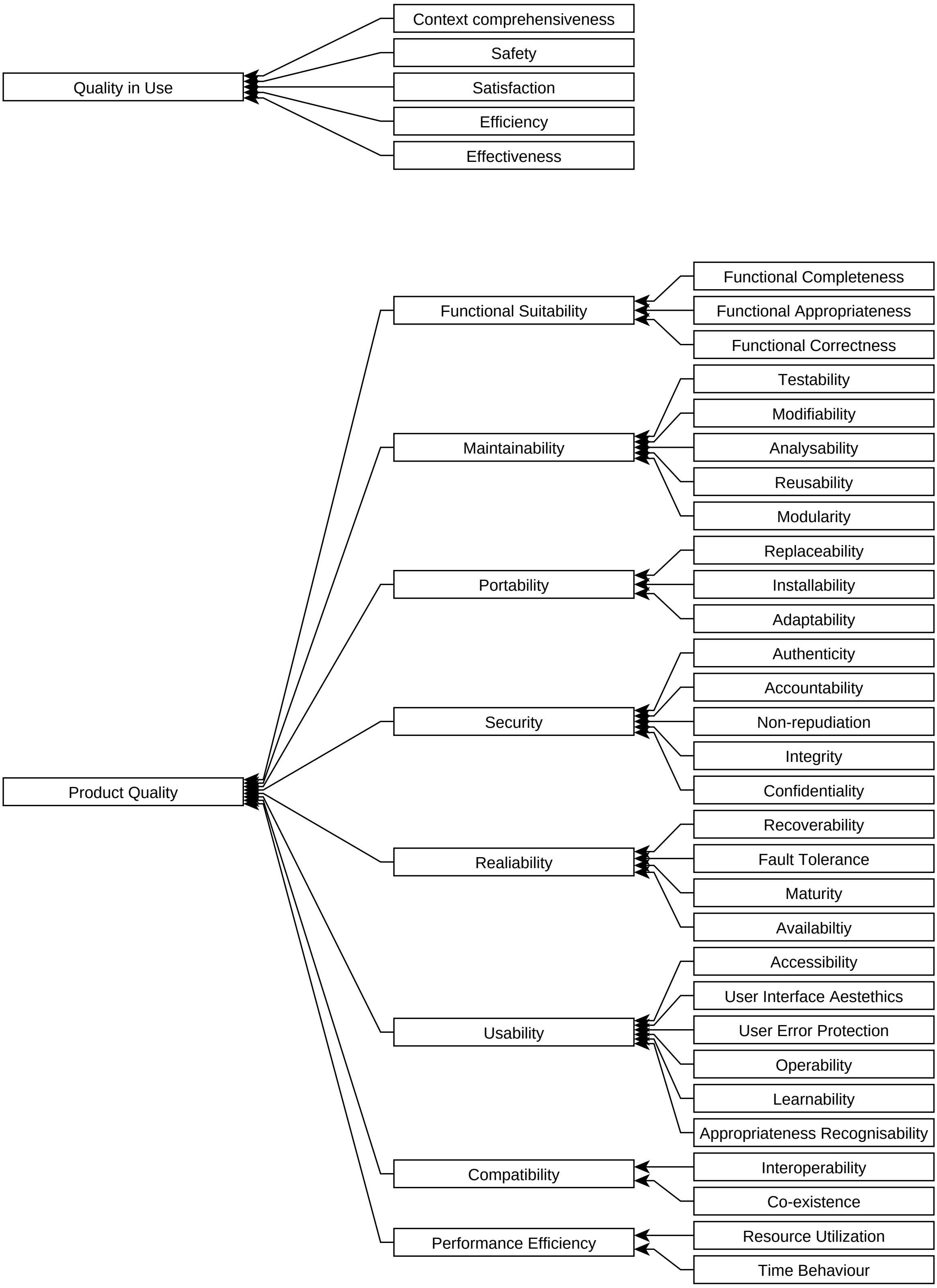


Figure 3.2: ISO 25010 Quality Model [65]

top level quality attribute. Moreover, the difference between *adaptability* and *maintainability* is unclear, because both refer to the modification of the product to satisfy certain needs.

- *Unclear Semantics:* The hierarchical models do not clearly define a semantic of their model [26, p. 37]. They even miss a meta-model defining the modeling elements used. This is one underlying problem leading to increased ambiguity.

- *Structuredness:* Deissenboeck [33] identified a fundamental structuring problem in the quality attribute hierarchies. They mix characteristics of the system's interactions with intrinsic product characteristics. For instance, *structuredness* and *analyzability* are both sub-characteristics of maintainability, even though structuredness is a characteristic of the product itself, while analyzability is defined as the "ease with which the impact of an intended change on the rest of the product can be assessed" [65], which is actually a characteristic of the activity of assessing the product. A consequent differentiation between activities and characteristics is the foundation of activity-based quality models [33].

- *Measurability:* The additions to the standard ISO 9126 (and also the measurement addition ISO 25021 [67] to the ISO 25010) propose indirect measures, relying "on observations of the interaction between the product and its environment" [58]. For example for measuring changeability, the "change implementation elapse time" is suggested as a measure. This is actually a process measure of the development process, not a product measure. Other measures are defined in a vague manner, such as the measures of "activity recording" for analyzability, which is defined as the ratio between the number of data items for which logging is implemented versus the number of data items for which logging is required. If there are no measures for a quality attribute available, the ISO 9126 suggests using some other related attributes and to predict the required characteristics. However, no approach for establishing the surrogate measures and the prediction system are provided [86].

- *Aggregation:* Even though unsatisfactory measures are proposed by the ISO 9126, no means for aggregating measurement results are provided [86]. Thus, there is no way of assessing higher-level quality attributes and thereby providing overall quality statements of software products. These problems have not been resolved in the newer ISO 25021.

3.1.2 Richer Models

After the publication of ISO 9126 researchers proposed more elaborate quality models, to overcome known shortcomings of the previous quality models. Dromey [34] introduced a distinction between *product components*, their *properties*, and externally visible *quality attributes*. As product components he uses the artifacts of the software or rather constructs of a programming language, such as *expressions*, *variables*, and *modules*. These components are attributed with properties, such as *computable*, *precise*, and *structured*. For each component property, such as *computability of expressions*, its influence on a quality attribute (taken from ISO 9126) is defined and explained by a rationale. Figure 3.3 shows an excerpt of the quality model for the product component *expression*. For this product component various component properties are defined. Each component property influences several quality attributes.

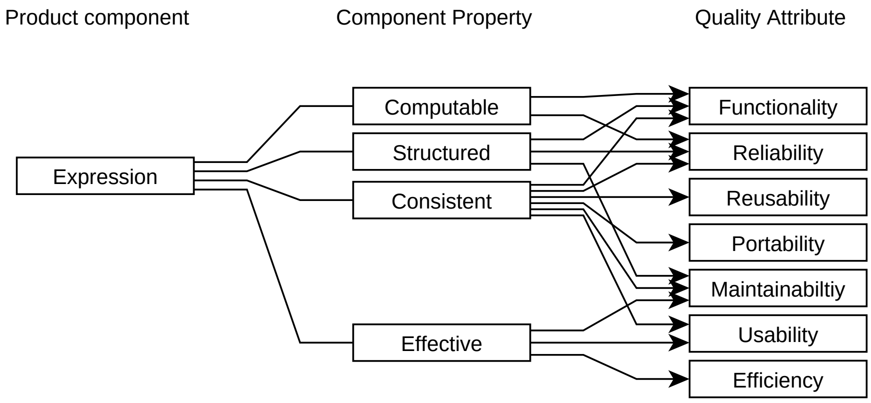


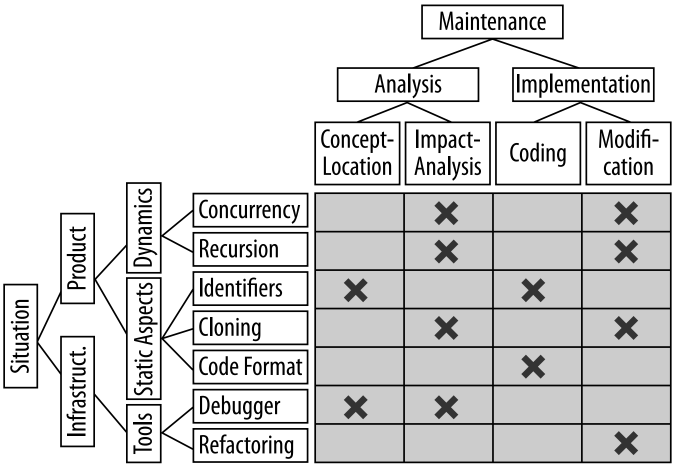
Figure 3.3: Dromey's Quality Model [34]

The structure of Dromey's model has been used by other authors to build their own quality models (compare discussion in [158]). Bansiya and Davis [6] built a quality model for object-oriented designs. They chose product components for object-oriented programming languages and described several measures for them. They developed tool support for automatic measurement and aggregation of measurement results.

A different approach for building useful quality models takes the SQUID approach of Kitchenham et al. [85]. They define an explicit meta-model for their quality models. Their meta-model defines hierarchical quality models and constructs in order to add measures and threshold values for measurement results. Based on the meta-model they define a "build your own" method for developing individual quality models for one's needs [158]. As a basis for defining quality models the quality attributes of ISO 9126 are used. To the low-level quality attributes measures are attached by the model developer. Furthermore, a target value for each measurement value must be defined to allow an actual quality assessment.

The EMISQ [122,123,129] method for building operationalized quality models focuses on assessing the source code by static code analysis. As a quality model they use a hierarchical quality model based on the ISO 9126. The measures are defined by rule-based static code analysis tools (e.g., PCLint, FindBugs), whereby each measure can be attached to one or more quality attributes. The quality assessment approach relies on both automatic aggregation and manual evaluation. For all rule-violations an expert has to provide a rating; the ratings are then automatically aggregated using the median. Tool support for building the quality model, defining measures, and for the automated quality assessments is provided. In [124] a concept for building a continuous quality monitoring and control approach and introducing it into organizations is provided.

The activity-based quality models of Deissenboeck et al. [26, 33] address the shortcomings of the hierarchical models regarding ambiguity, structuredness, overlapping and unclear semantics by introducing a new concept for describing quality. Instead of defining arbitrary quality attributes, they describe quality in terms of activities conducted with or on the system. For instance, instead of talking about *maintainability*, *analyzability*, and *modifiability*, they define the activities *maintenance*, *analysis*, and *implementation*. Activities provide a clear semantics and a clear decomposition criteria. For instance, it is clear that maintaining a software means analyzing a change request in the given system, implementing the requested modifications, testing, etc. In addition to the activities,



	Concept-Location	Impact-Analysis	Coding	Modification
Concurrency		✗		✗
Recursion		✗		✗
Identifiers	✗		✗	
Cloning		✗		✗
Code Format			✗	
Debugger	✗	✗		
Refactoring				✗

Figure 3.4: Activity-based Quality Model [33]

inherent characteristics of the product are described in this model. Then, the model describes the influence of product characteristics on the activities and gives a rationale for each influence in prose. Figure 3.4 shows an example for the activity *maintenance*. We can see that the cloning has an influence on the activities *impact analysis* and *modification*. Further activity-based quality models, describing the quality attributes *usability* [166] and *security* [105, 163], have been proposed.

Squale [112, 113, 142] acknowledged the problem of directly connecting measures of quality attributes of ISO 9126. Therefore, they introduced an additional layer, called *practices*. Practices describe technical principles that should be followed by developers. Examples of practices are that the *class cohesion* should be high, the *number of methods* in each class should not be too high, and that the *encapsulation* of members of classes should be high. For developing the Squale quality model, a bottom-up approach was chosen. The practices have been derived from existing (and mostly automatically calculable) measures. For each practice an aggregation function is specified, so that the entire model is operationalized and automatic quality assessments are possible. For the aggregation they use expert-based threshold values. For classical size and complexity measures (e.g., size of methods, number of methods per class, nesting depth of methods) an expert defines a function mapping to the interval $[0, 3]$, zero meaning a bad result and three a good result. As aggregation functions they adapted econometric inequality indices. For instance, the size of methods is mapped to this scale by the function $2^{\frac{70-SLOC}{21}}$. Obviously, the values 70 and 21 are based on expert knowledge of the sizes of methods. For rule-based static code analysis measures a similar transformation is used, essentially being based on a defect density measure: $a^{\frac{w \times \text{numberofdefects}}{SLOC}}$. The constant a and the weight w are chosen by the expert. Based on the quality model, they provide tool support for evaluating software products. The measurements and the quality model are fixed within the tools.

Critique

The richer models mostly focus on single problems of the hierarchical models and improve them. The model given by Dromey first introduces product components in order to discuss separately the components of the product and their characteristics. However, he does not address the issue of measurement and aggregation. The SQUID model of Kitchenham et al. first introduces an explicit meta-model for quality models. This way, they address the problem of unclear semantics of the model elements. The overall structure of the resulting quality models is nonetheless equal to ISO 9126, with all the problems known there. Besides directly assigning measures to quality attributes and defining thresholds, the challenge of conducting quality assessments is not addressed. EMISQ takes a bottom-up approach and focuses on building quality models for automated quality assessments based on rule-based static code analysis. As quality attributes, the ISO 9126 is used and no advances regarding the structure of quality characteristics are made. The activity-based quality models focus on the challenge of structuring and unambiguously defining quality characteristics. However, they do not address the problem of measurement and aggregation. Squale focuses on the challenge of quantifying quality attributes by measures. Although aggregation of measurement results is improved by this model, the problem of structuring quality attributes is not addressed.

Despite the quality models discussed here address single issues of the hierarchical models, a comprehensive model solving all issues is still missing. Both the main challenges, the definition of quality attributes and the operationalization of a quality model, are still unsolved. For instance, the activity-based quality models provide a clear way of defining quality attributes, however, they have so far only been applied to maintainability, usability, and security. Another example is the EMISQ model, which focuses on quality assessments, but has a narrow focus on static code analysis and expert ratings.

3.2 Software Measurement

In science and engineering it is essential to measure attributes of things. Measuring is the base for formulating hypotheses and testing them, and it is the foundation for setting goals and measuring their achievement [36, p. 9]. The discipline of software measurement (often called software metrics) is about rigorously applying measurement to software engineering.

In this discipline, a large number of measures for quantifying several attributes of software emerged. We discuss the most prevalent ones in the first subsection. Then, we discuss the well-known measurement framework, GQM. At last, we take a look at common tools for calculating software measures and reporting them.

3.2.1 Software Measures

One of the first measures that emerged in software engineering were measures for the size of software. A possible measure for the size is the length of the source code in lines (LOC). Over time, different variants of this measure emerged, taking into account that code also contains blank lines and comment lines. Thus, a measure counting non-commented lines of code (NCLOC) emerged and is widely used [36, p. 247]. Other similar measures are “number of statements”, “number of functions”, “number of classes”, etc.

Often the size of the code is considered a misleading measure for the size of the system. Rather, a measure for the amount of functionality is proposed. The first such measure is *function* *points* [36, p. 259]. They are calculated on a specification by counting external inputs, external outputs, external files, and internal files. Each of these counts is weighted by a complexity factor and its sum constitutes the function point measure. This measure has been fiercely criticized [36, p. 263] chiefly for subjectivity and technology dependence. One fundamental problem of measuring functionality is that for a given problem there may be several solutions of different complexity. It cannot be guaranteed that a solution chosen in a program is the least complex one. Thus, measuring the complexity of the problem with function points may not represent the complexity of the given program [36, p. 267].

To measure the complexity of a piece of source code, complexity measures have been introduced, which actually measure structural attributes of the source code. They refer to either the control flow, data flow, or the data structure itself. One of the best known measures of this type is the cyclomatic complexity measure of McCabe [108]. It measures the linearly independent paths through the flow graph of a given program. Other measures of a similar type are, for instance, "Halstead Volume" [55], or simpler ones such as nesting depth.

An objective often pursued with measures is to predict faulty components of software systems. A large number of studies experiment with different measures and prediction algorithms [52, 82, 97, 115]. Hall et al. [54] give a comprehensive overview of the state-of-the-art in fault prediction. In general, predictors build for one software system (or for similar systems) are not transferable between different systems. For fault prediction based on product measures the results are mixed. In some studies, code complexity measures perform well, while in others they do not. Generally, a combination of process measures and product measures performs best. Lincke et al. [97] apply different quality prediction models to one set of software systems and come to the conclusion that the different prediction models yield inconsistent results for the systems under evaluation.

Critique

Regarding the usage of software measures for quality assessments there are three major critiques:

1. Despite the large number of software measures, it is still unclear how they relate to quality in general or to specific quality attributes. Hence, their gainful application in quality assurance is still unclear.

2. Additionally, measures are often defined based on available data, instead of actual measurement goals [26]. Thus, most measures are defined on source code, because source code is usually available and easy accessible.

3. A third topic is the validity of software measures. Generally, the validity depends on the objective pursued with the measure. As we have seen, for the prediction of faults, software measures are of limited use. Though fault prediction is just one single topic in the wide area of quality assessments. In general, for assessing the quality of software, it must be assured that the measures conform to the measurement goal. To achieve conformance, measurement frameworks have been introduced. The most prominent of them, the Goal-Question-Metric, is discussed in the following subsection.

3.2.2 Measurement Frameworks

A widely known and used measurement framework is the Goal-Question-Metric (GQM). It was first published in 1984 by Basili et al. [10] and then further developed by Basili and Rombach [8]. The authors observed that often measurement in software development was applied without a clear goal. Measurement projects often failed because only easily available measures were collected, instead of useful measures with regard to a specific project goal. To alleviate this problem, they proposed a top-down approach for defining measures:

1. Define company-wide *goals* (or department specific goals).
2. Derive from each goal the *questions* that must be asked to assess whether the goal was achieved.
3. Define *metrics* for each question to quantify it.

By this strict top-down approach, GQM is able to select meaningful measures. Furthermore, it avoids a too large number of measures, which would then be hardly manageable if selected. GQM has often been applied in an industrial context [12, 153, 154]. However, it is mostly used for process assessments using process measures, and not for product assessments.

Critique

The measurement frameworks clearly address the narrow scope of measures, by defining a top-down approach. Hence, they ensure that actually important and/or interesting characteristics are measured and not just easily available ones. However, their guidance is at a very high level of abstraction, leading to a "do-it-yourself" quality model approach [26, p. 54].

Regarding the product assessment with GQM, it is criticized that constructing the tree of goals, questions, and measures leads to a similar structure such as that of hierarchical quality models [26, p. 53]. Thus, it faces the same problems as the hierarchical models and the SQUID quality model approach discussed in Section 3.1.2.

3.2.3 Tools

A large number of tools for automated quality analysis of software systems have been developed. In the literature, different categorizations and collections of such tools have been published (see [31, 158, 159]). We use the categorization into *analysis tools* and *dashboard tools* and summarize and complement existing discussions of the literature in the following.

Analysis tools have the objective of executing measures and reporting the measurement results. One group of such tools are *rule-based static code analysis tools*. They search for certain patterns in the source code, by analyzing, for instance, the data and control flow. They report locations in the source code, where a rule is violated, which indicates a possible error. Examples for such tools are FindBugs [37] and PMD [125] for Java, PCLint [120] for C/C++, and Gendarme [111] for C#. A similar type of tools are the *coding convention checkers*. They check the visual alignment of the source code and report anomalies or reformat the code automatically. Examples for such tools are Checkstyle [20] and astyle [3] for C/C++, C#, and Java. Many development environments and editors, such as Eclipse [146] support the formatting of source code. A third group of tools also operating on source code conduct measurements of classical software measures, such as size and complexity measures. The execution of such measurements is often integrated in dashboard tools or in tools providing visualization support, for instance, Understand [149]. Besides classical software measures, tools for specialized analyses of source code are also available, for instance, the CloneDetective [78] for identifying duplicated code. More specialized tasks include architecture conformance analyses. These rely on a predefined target-architecture and assess a software system for violations of this architecture. Examples of such tools are the architecture analysis part of ConQAT [28] and JavaDepend [75].

Dashboard tools have the objective of providing an overview of the quality of a software system to developers, quality assurance staff, and managers [31]. Thus, the visualization of the measurement results is a primary objective of them. For the different target audiences they provide especially adapted views. Managers are first and foremost interested in a broad overview of the quality data and on trends in order to judge whether the quality is improving or deteriorating. Developers need additional views to trace high-level results to single components and locations in the source code, in order to correct quality deficits. The dashboard tools often rely on the analysis tools described in the previous paragraph. An example of a tool which focuses primarily on integrating existing analysis tools and providing visualizations and trend data is ConQAT [30]. Other examples of such tools are QALab [169], Sonar [140], and XRadar [93].

In addition to the tools mentioned above, there are some experimental research tools. Marinescu et al. [106] and Schackmann et al. [134] take a first step to integrating an explicit quality model and an assessment toolkit.

Critique

The analysis tools are generally an implementation of the measures discussed in Section 3.2.1. Thus, they do not advance on the state-of-the-art discussed in that section.

The dashboard tools have the broader objective of providing an integrated view on software quality. To do so, they provide trends of measurement values over time and aggregate measurement values to quality indicators. As a backbone for the aggregation, they usually use the hierarchical quality models. Hence, the resulting aggregations face the same problems as the quality models themselves do. Due to the unclear decomposition of quality characteristics, the aggregated values are hardly comprehensible and hard to explain to practitioners.

3.3 Definition of Quality Requirements

A precondition for all constructive and analytic quality assurance activities is to define the target quality for the product. Defining the target quality means that specific quality requirements have to be defined. This is a classical task of requirements engineering. In requirements engineering two techniques for elicitation are typically used: (1) scenario-oriented techniques and (2) goal-oriented techniques.

Scenario-oriented techniques focus on capturing the interaction between end users and other stakeholders with the system. They have been proven a suitable technique for interaction with stakeholders, because thinking of exemplary interaction sequences is a natural form of communication for people [74]. Furthermore, by using scenarios, the interaction between user and developer is focused on the *use* of the future system by describing *what* people can do with the system and what the consequences are for the people and their organization. By describing the tasks the user is conducting with the system, they explain *why* a system is needed. In software engineering scenarios are widely used in the form of use cases [73] for specifying primarily functional requirements. Yet, there are different extensions to use cases for capturing non-functional requirements with them too [137].

Another set of techniques for capturing quality requirements found in literature are goal-oriented approaches (see, e.g., [151]). Those techniques start from goals of stakeholders and refine the goals till they get to concrete requirements. Due to the general nature of a goal, they are also suited to formulating and reasoning about the required quality characteristics of a product. There exist many variations of this goal-oriented paradigm, ranging from informal approaches (see, e.g., [80, 118]) to formal approaches (see, e.g., [24, 25, 95, 152]).

Both the scenario- and goal-oriented approaches offer a methodology for eliciting quality requirements, but they give no advice on the structuring and specification of quality requirements. Moreover, they give no advice for the different quality-related topics that have to be captured in requirements elicitation. Another problem is that these approaches do not guarantee that the derived requirements are concrete enough for being useful in constructive and analytic quality assurance.

A third type of approaches addresses these problems by providing more guidance in form of a quality model (see, e.g., Kitchenham et al. [85]). However, they only use a taxonomic quality model like ISO 9126 [70] and are thus not able to tackle the major shortcomings discussed above.

3.4 Quality Models in Practice

In this section, we present an analysis of the use of quality models in the software industry that is based on a broad Web survey conducted in the context of the Quamoco project. These results have been published in [161, 162]. The main objectives of the study are to identify the main classes of quality models that are used at present, to identify the quality assurance techniques that are applied with these quality models, and to identify problems that are related to these models. The study covers different company sizes and several domains to allow a broad view of software product quality.

3.4.1 Study Definition

We concretize the study using the Goal-Question-Metric goal template as proposed by Wohlin et al. [167]: Analyze *product quality models* for the purpose of *characterization* with respect to their *usage (where and for what)* and *possible improvement potentials* from the viewpoint of *quality managers* and *quality model users* in the context of the *software developing industry*.

3.4.2 Study Design

Before conducting the actual study, an extensive pre-study has been done. In the pre-study, face-to-face interviews with employees of industrial participants of the Quamoco project were carried out. The interview guidelines and questionnaire were developed in a series of workshops. The final version of the questionnaire contained 15 open questions and 12 closed questions. Detailed information about the pre-study was published separately in [161].

The actual study comprises a survey using a Web-based questionnaire. The questionnaire was developed based on the closed questions of the pre-study and the data and experience gained in the pre-study. The Web-based questionnaire was initially tested internally with employees of the Quamoco industrial partners to detect potential weaknesses and to estimate the time required for completing it. Afterwards, the questionnaire was further refined to eliminate the problems detected in the internal tests.

The final version of the Internet-based questionnaire consists of 23 questions that are divided into the four groups:

1. The role of quality models in the company
2. Quality requirements and evaluation of quality
3. Improvement of quality models
4. General information about the company

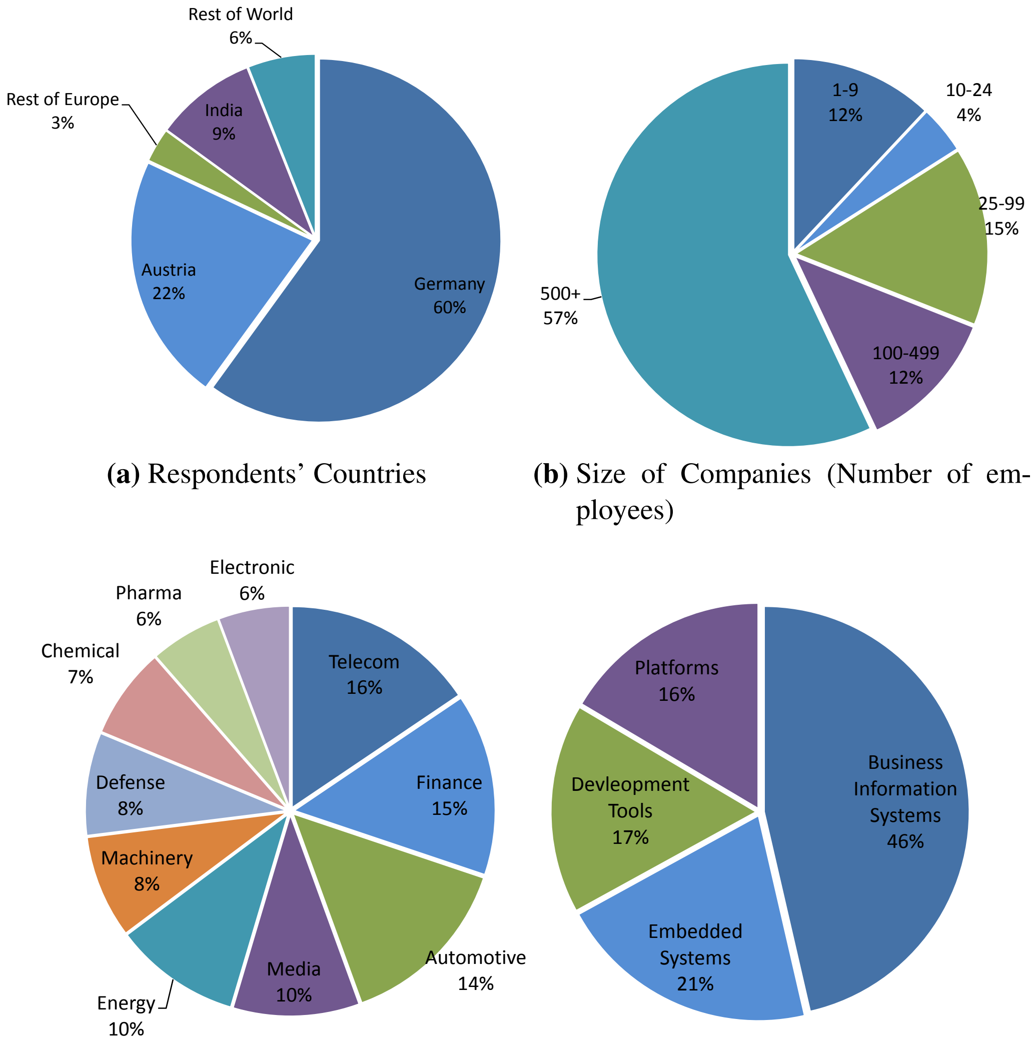
The last question of each group was an optional open question, which allowed the respondents to state additional comments of any kind. Based on the experience of the pre-test, we estimated a time of at most 25 minutes for completing the questionnaire. For the final survey, the questionnaire was implemented in LimeSurvey [96], a free Web survey tool.

3.4.3 Study Subjects

The population of this study consists of quality managers and quality model users employed at software development organizations distributed over the whole world. The recipients of the Internet-based questionnaire were selected using convenience sampling. Based on data from the project partners, a list of recipients was compiled. This selection resulted in a sample of 515 persons. We contacted the persons from the sample, out of which 125 completed the questionnaire. This corresponds to a response rate of 24.3%. Although we used personalized contacts and sent one follow-up e-mail to those who had not answered, we did not establish pre-contacts.

Figure 3.5a shows the countries the respondents came from. The majority (82%) of the respondents work in Germany and Austria. Hence, we have an emphasis on German-speaking countries. This is complemented by 18% participants from other European countries, Asia, Africa, and North America. Forty-four percent of the respondents work as project managers. Twenty-eight percent of the respondents are normal employees or line managers.

The experience of the respondents is important to ensure that the respondents are able to answer the questionnaire meaningfully. Twenty-nine percent of the respondents have between 11 and 15 years



(a) Respondents' Countries

(b) Size of Companies (Number of employees)

(c) Industry of the Participants

(d) Type of Software

Figure 3.5: General Information about the Companies

of professional experience in the area of software engineering; only 15% of the respondents have less than five years professional experience.

The majority of the respondents see development as the primary task of their department. More than a third consider their department mainly in quality assurance. Figure 3.5b shows the size of the organizations in terms of the number of employees. Though the participating organizations cover all sizes, the emphasis lies clearly on larger companies.

Figure 3.5d shows the types of software developed by the participants of the study. It covers all major types of software (business information systems, software for embedded systems, development tools, and platforms). This software is used in all major domains ranging from telecommunication to finance (see Figure 3.5c).

3.4.4 Study Results – The Role of Quality Models

Which quality models/standards do you use for assuring the quality of your products?

The participants were asked to answer one or more of the options “ISO 9126”, “ISO 25000”, “domain-specific”, “company-specific”, “laws”, “quality gates”, “defect classification”, “reliability growth models”, “none”, or “other”. In case of domain-specific or other models, the participants could give the name of the model as free text. Figure 3.6 shows the results as the number of answers per option.

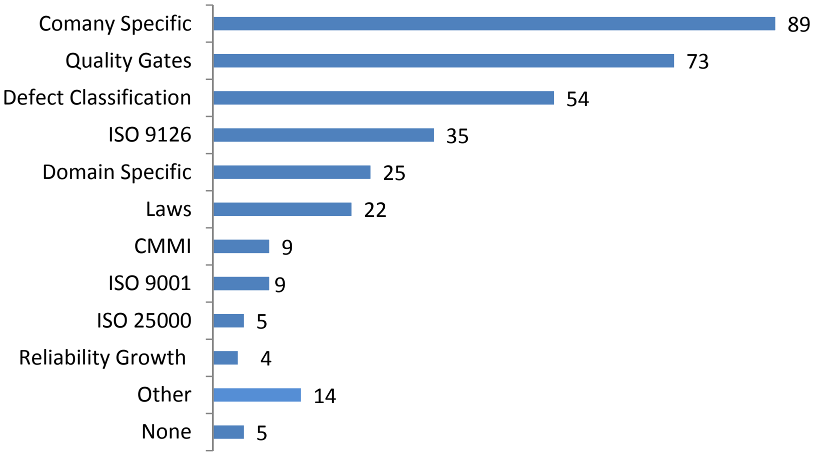


Figure 3.6: Quality Models Used in Practice

The predominant kind of quality models that are used in practice are company-specific models. Almost three quarters of the respondents use this type of quality model. Well-established practices also include quality gates and defect classifications, with about half of the participants employing them. As these terms are rather vague, the actually used models can, however, differ to a high degree. In the first study phase, we found that defect classifications are often only a prioritization of defects.

The data also suggests that the standard ISO models are not highly accepted. ISO 9126 is adopted by less than a third of the participants and ISO 25000 by only 4%. For the latter, a reason might be the recent and incomplete publication of the standard at the time the survey was conducted.

From the pre-study, we found that the ISO models are mostly used in combination with company-specific models. Hence, we expect this to be the case in the larger sample as well. To test this, we state the following hypothesis:

Hypothesis H_1: In addition to the ISO 9126 model, company-specific models are used.

The contingency table in Table 3.1 shows the co-occurrences of using and not using company-specific models as well as the ISO 9126 model. It shows that the ISO 9126 is used 2.5 times more often in conjunction with company-specific models than without. Hence, we accept the hypothesis H_1.

Domain-specific quality models and laws were also mentioned frequently. The mentioned domain-specific models originated from the area of pharmaceutics and medicine (ISO 13485, IEC 60601, IEC 62304, MDD, CFR) with 11 occurrences in total, testing (4), safety (4), security (3), public or military (3), and accessibility (2). Five respondents answered that they use the new ISO 25000 and

		Company-Specific Quality Model	
		Not used	Used
ISO 9126	Not used	26	64
	Used	10	25

Table 3.1: Contingency table for ISO 9126 and company-specific models

only four employ reliability growth models. Five respondents (4%) answered that they do not use quality models.

For this question a sanity check was performed, i.e., “none” was not supposed to appear together with any other field. The check was successful for all 125 responses. In addition, the answers for “other” led us to corrections in four cases where it was obvious that the mentioned models belong to one of the predefined categories. The remaining models in “other” were mainly “CMMI” and “ISO 9001”, with nine occurrences each. This is why we introduced them as separate categories and removed them from the “other” category.

Do you adapt the quality models that you use for your products?

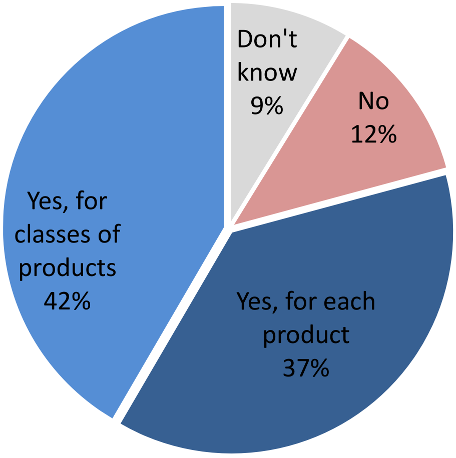


Figure 3.7: Adaptation of Quality Models

The participants had three options for answering this question: They do not adapt the used quality models (“no”); they adapt them “for classes of products”; or they adapt them “for each product” separately. Furthermore, a “don’t know” answer was possible. Figure 3.7 shows the relative distribution of the answers including the “don’t know” answers. Quality models are commonly adapted. Ignoring the “don’t know” answers, 13.2% do not adapt their quality models. The difference between the adaptation for classes of products and single products is only four percentage points; thus, both kinds of adaptation are likely to happen in practice. We conclude from this that adaptation is a necessary task in using quality models independently of whether it is employed for a single product or for classes of products. It might even be an indicator that the existing standards are not sufficient for practical needs.

In detail, of the 125 responses, 11 checked “don’t know”. Therefore, only 12.0% do not adapt their quality models. The quality models of 79.2% of the participants are adapted, of which 41.6% are adapted for classes of products and 37.6% for each product.

How satisfied are you with the quality models you use for your products?

The respondents could answer on a 10-point ordinal scale from “very satisfied” to “very dissatisfied”. Alternatively, a “don’t know” answer was possible. Figure 3.8 shows the distribution of the answers on this scale including the absolute number of answers.

The average satisfaction with the quality models used tends to an undecided opinion with a slight shift in the direction of satisfaction. The quality model users seem not be completely unsatisfied with their models. However, a clear satisfaction is not observable either. This suggests that the concept of quality models seems to deliver a basic level of satisfaction but it has still room for improvement.

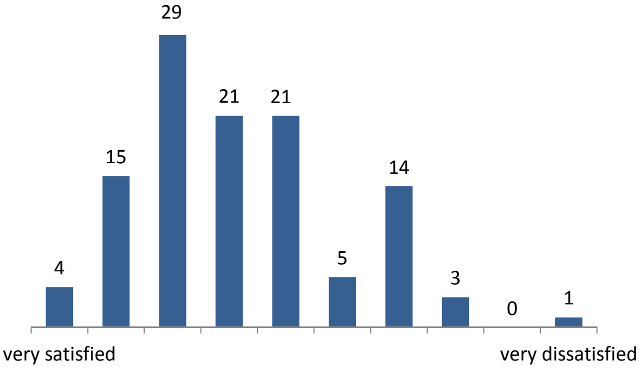


Figure 3.8: Satisfaction with Quality Models

Eight of the values are missing; thus, 117 values remained for further analysis. The answers are coded by means of a variable with a scale from 1-10, where one codes “very satisfied” and 10 “very dissatisfied”. The whole range of values from one to 10 was used by the respondents. The median is four, the mean is 4.21 with a variance of 3.480.

In which development activities do you use quality models?

These activities were asked for: requirements engineering, architecture and design, development of coding guidelines, informal reviews, formal reviews, tool-based code analysis, data collection and measurement, testing, and evaluation of customer feedback.

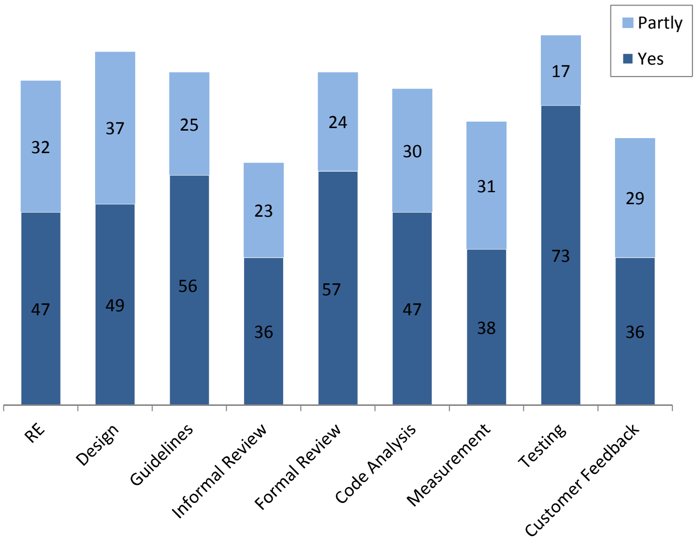


Figure 3.9: Quality Assurance Techniques that Use Quality Models

For each of these activities the respondents could give one of the following answers: the activity is not part of the development process (“activity does not exist”); quality models are used in this activity (“yes”); quality models are only partially applied (“partly”); or quality models are not used (“no”). Otherwise, the respondent could state a “don’t know” answer. In addition, a free text question, gives the respondents the opportunity to state, for instance, the motivation why they use a quality model in a development activity. Figure 3.9 depicts the relative number of "yes" and "partly" answers for each activity.

Quality models are frequently used in all development activities. In particular, the testing activity is supported by quality models and, additionally, quality models are used in a thorough manner. This is reflected by the low number of participants who answered "partly" (7%). This is in contrast to architecture and design, in which quality models are often used partially (37%). The slight dominance of testing may be explained by the important role testing plays in quality assurance in general (see next question).

For each activity, the respondents who answered "does not exist" or "don't know" were excluded. The three activities in which quality models are used most are testing (73% of the respondents answered "yes"), formal reviews (57%), and coding guidelines (56%). The three activities in which quality models are used least are informal reviews (22% of the respondents answered "no"), customer feedback (16%) and data collection (14%).

How important are the following quality attributes for your products?

For each quality attribute a 10-point scale from "very important" to "very unimportant" is given. Alternatively, for each quality attribute, a "don't know" answer was possible. Figure 3.10 visualizes the central tendency and range for each quality attribute. The boxes represent the values from the

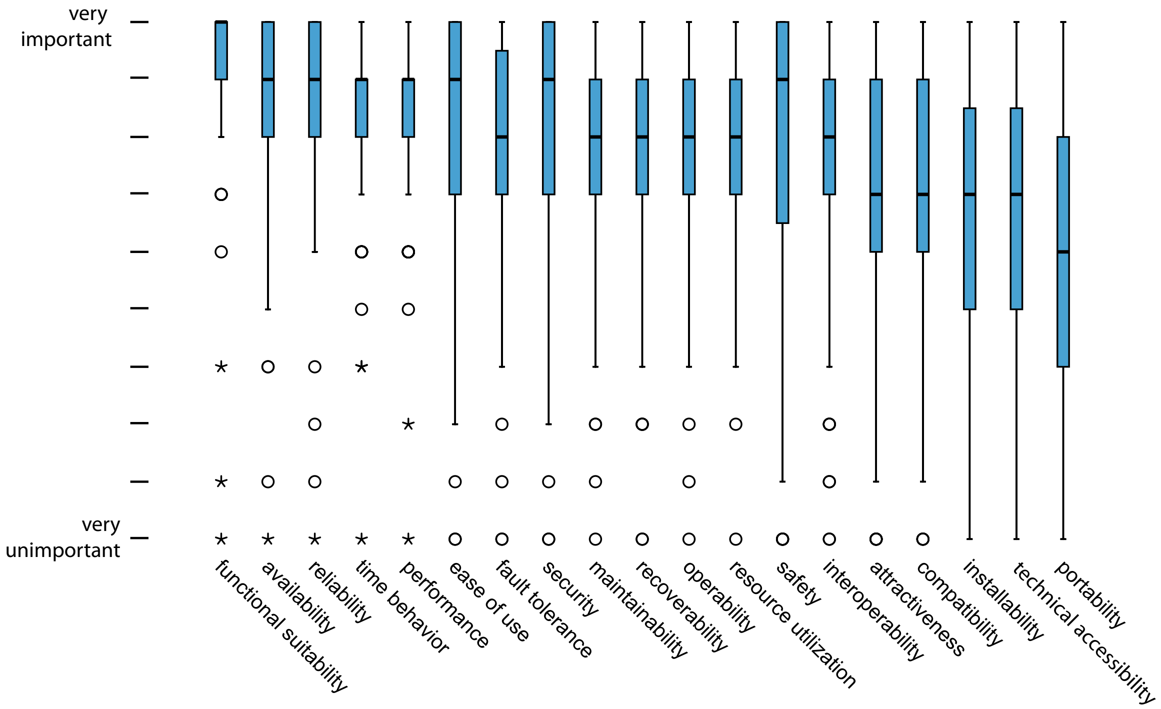


Figure 3.10: Importance of Quality Attributes

lower to the upper quartile and the horizontal line marks the median. The upper and the lower whisker represent the minimum and the maximum; the circles and asterisks visualize outliers.

The analysis shows that the individual ranking of quality attributes varies and that the importance of quality attributes does not differ strongly. The former can be seen in the ranges of the quality attributes that use the whole spectrum of possible answers for all of the quality attributes. The latter is derived from the small distribution of medians, of which most are seven or eight. As medians, the ranking "very important" (10) and "medium important" (6) only appear once. Hence, the distinction of importance between the attributes is not clear. However, functional suitability tends to be the most important quality attribute whereas portability seems to be the least important one. We conclude that all quality attributes are important. Depending on the context, each attribute can be of most importance. Nevertheless, foremost in participants' minds is the functional suitability of their software together with its reliability and performance. Portability and installability are only of high relevance in specific contexts, probably standard software. The "don't know" answers are counted as missing. The numbers of missing values range from three to nine; thus, 116 to 122 answers are available for each quality attribute.

3.4.5 Study Results – Quality Requirements and Evaluation

How important do you rate the following techniques for evaluating quality requirements?

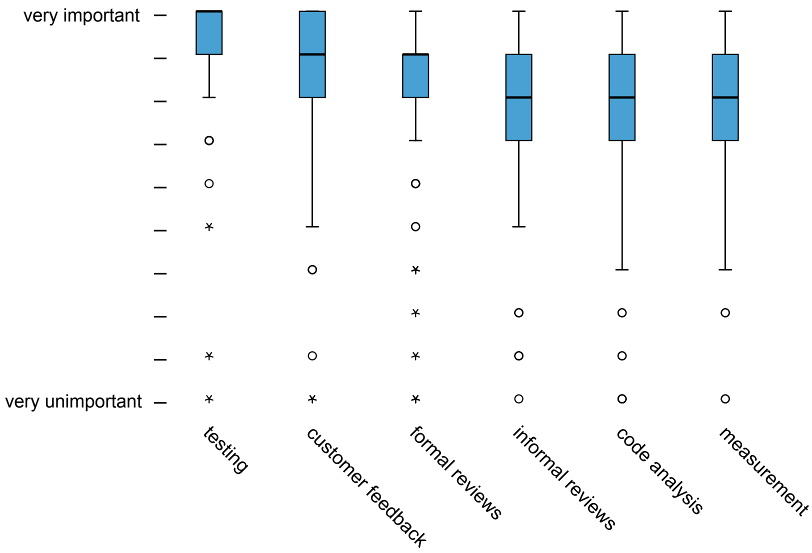


Figure 3.11: Importance of Quality Assurance Techniques

The following techniques were considered: informal reviews, formal reviews, tool-based analyses, data collection and measurement, testing, and evaluation of customer feedback. Each of these techniques could be rated from "very important" to "very unimportant". If the respondents could not give an answer, they could check the "don't know" option. Figure 3.11 shows the distributions of the importance ratings. For each of the techniques the median is marked by a black line. The whiskers give the quartiles and circles and asterisks denote outliers.

From the data, we conclude that testing is considered the most important technique for evaluating quality requirements. The techniques that were rated as most important after testing are formal reviews and customer feedback. The importance of a technique does not necessarily mean that quality models are used accompanying. This is shown by the technique customer feedback: The use of quality models for this activity is in the minority (36%), although the activity is considered important.

Looking at the distribution of the medians, similarly to the quality attributes, the importance does not differ strongly. The range of the medians contains only three of 10 scale points. Hence, overall, all the mentioned techniques are considered important.

The “don’t know” responses were not considered further. However, the number of these responses was very low for most techniques (0-2), except for the technique ”measurement” (11).

How often do you use the following quality evaluation techniques?

The same techniques as in the previous question were used. For each of these techniques the following answers could be given: “daily”; “weekly”; “monthly”; “at specific milestones”; and “never”. It was also possible to answer “don’t know”. For this question, multiple answers were allowed. That means it was possible to answer that an activity is performed once a month and also at certain milestones. Figure 3.12 shows the relative values in a stacked bar chart.

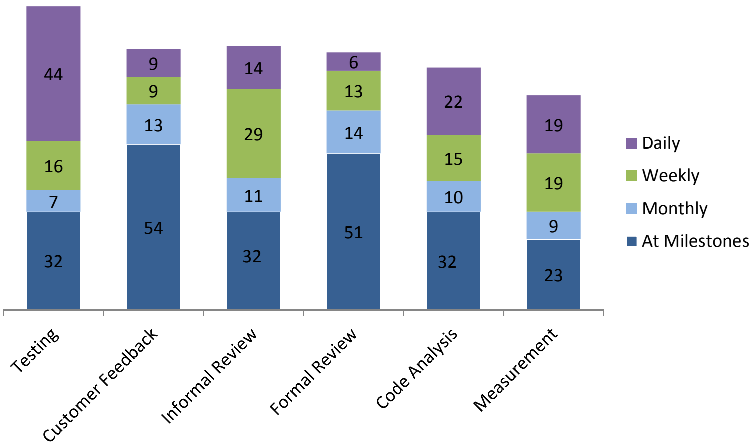


Figure 3.12: Frequency of Quality Assurance Techniques

Techniques that can be carried out fully automatically, such as testing, are in favor when it comes to evaluation on a daily basis. Other techniques that are more time consuming yet important, such as formal reviews, are used predominantly at milestones. Monthly is not a very common interval for quality assurance. Likewise “weekly” is only popular for informal reviews.

The two techniques that obtained the most answers of “never” were formal review (27%) and code analysis (26%). The two techniques that are used most often at milestones are customer feedback (23%) and formal review (23%). The two techniques that are used most often in a monthly interval are formal reviews (22%) and customer feedback (20%). The two techniques that are used most often on a weekly basis are informal review (29%) and data measurement (19%). The two techniques that are used most often on a daily basis are testing (41%) and code analysis (19%).

For this question, a sanity check was performed to make sure that “never” as well as “don’t know” did not occur together with any other option. In two cases this test failed. The corresponding inconsistent answers were considered missing as well as the “don’t know” answers. These answers were not taken into consideration for the further analysis.

How do you evaluate the specific quality attributes?

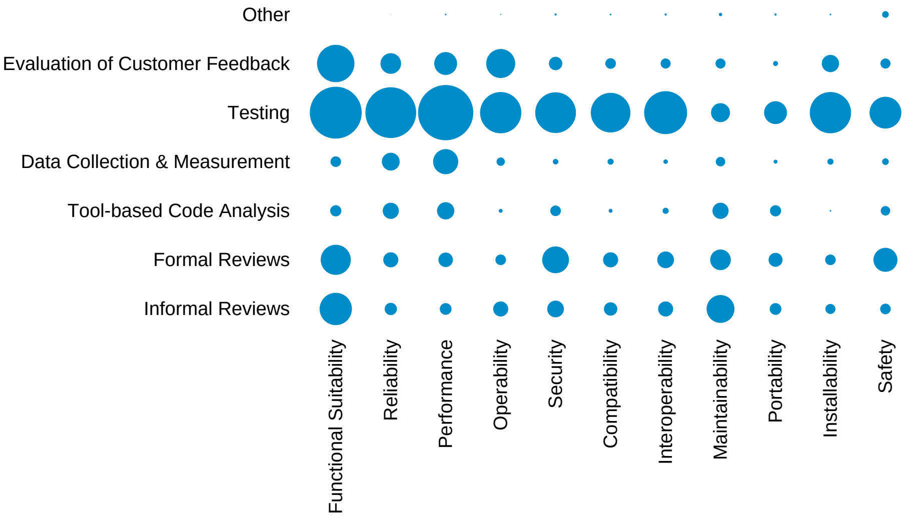


Figure 3.13: Quality Assurance Techniques Per Attribute

The answers are designed as a matrix with the specific quality attributes on the vertical and the evaluation alternatives on the horizontal level. For this question, multiple answers are possible. Besides the usage of a technique for a quality attribute, the participants could also state “don’t evaluate” and “don’t know”. The results of this question are shown in Figure 3.13. The larger the bubble, the more often the corresponding combination of quality attribute and technique was checked.

For this question, two main results are relevant: Functional suitability is the quality attribute evaluated most intensively and testing is a very important evaluation technique for all quality attributes except portability and maintainability. In addition, the results show that customer feedback is relevant for functional suitability, reliability, performance, and operability. Furthermore, measurement as well as tool-based code analysis are rarely used for any quality attribute with the exception of reliability and performance.

The combination of performance and testing was mentioned most often. Of the respondents, 79% use testing to analyze the performance of their software systems. The second and third highest frequencies are testing for functional suitability (74%) and testing for reliability (73%). The lowest frequency is the combination tool-based code analysis and installability with 2%. Portability analyzed by tool-based code analysis (6%) and operability and tool-based code analysis (6%) are mentioned similarly infrequently. In general tool-based code analysis (193 answers in total, 14%) and data collection & measurement (187, 14%) are mentioned least frequently. Informal review (317, 23%), customer feedback (334, 24%), and formal review (355, 26%) are mentioned similarly overall. Testing is mentioned by far the most often with 784 (57%).

Do you aggregate the quality evaluation results into an overall quality statement using a quality model?

A description of the “overall quality statement” is provided in the questionnaire to ensure that all respondents have a common understanding of this term. The possible answer categories for this question are “yes”, “no” and “don’t know”.

For the analysis, all mentions of “don’t know” are considered as missing values. The results of this question are shown in Figure 3.14.

According to the findings, there is no tendency in favor of or against the aggregation into an overall quality statement. For the interpretation of this question, we take questions allowing additional textual comments into account. One respondent who aggregates the quality evaluation results states that the aggregation is made for the understanding of the management. Other respondents note that the aggregation is used for visualization. Opinions against the aggregation are that the data for the aggregation is not available. Suitability for daily use is another reason against aggregation, as a respondent commented. We cannot clarify completely which reasons are responsible for or against an aggregation. For further analyses it may be expedient to ask explicitly for that.

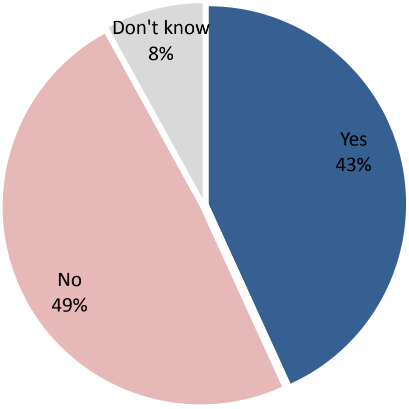


Figure 3.14: Overall Quality Statement

Ten respondents were not able to answer this question within the two categories “yes” or “no”. For this reason, the number of valid values is reduced to 115. The frequencies reveal that there is no major difference between “yes” and “no”.

3.4.6 Study Results – Improvement of Quality Models

Which improvement potentials do you see in the following fields?

On a 10-point scale from “very low potential” to “very high potential”, the respondents had to specify how they estimate the prospects of improvement suggestions. The respondents who did not require a certain improvement could choose a separate category. The fields for improvement given were: “defining practically usable quality models”; “adapting quality models to specific application contexts”; “integrating quality models into lifecycle processes”; “transparent definition of qual-

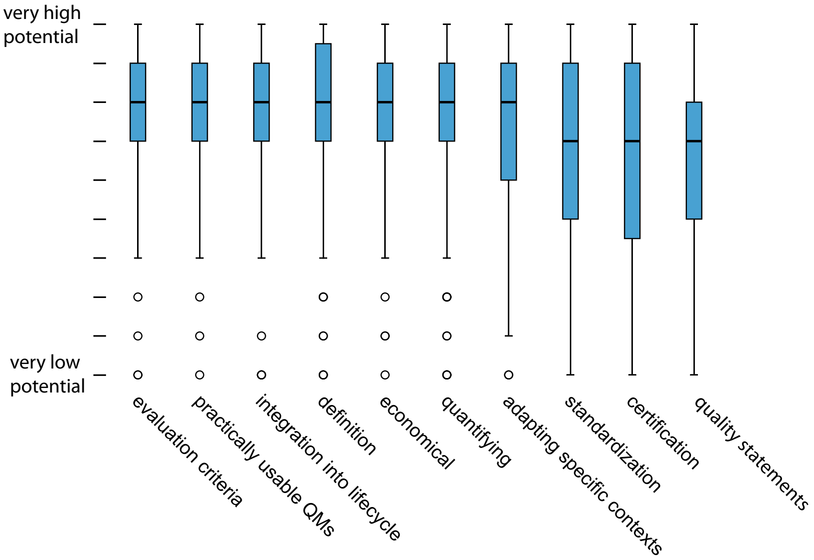


Figure 3.15: Improvement Potentials for Quality Models

ity"; "quantifying quality"; "defining evaluation criteria"; "economical analyses of quality improvements"; "standardization of quality models"; "aggregation to quality statements"; and "product quality certification". Figure 3.15 shows the distribution of the answers for each improvement field.

The findings show that for all fields a high improvement potential exists. More advanced topics, such as quality statements, standardization, and certification, may be rated with lower potential due to issues besides the use of quality models. For instance, standardization has the likely problem that it takes a long time to complete the standard and that many different interests have to be incorporated. Looking at the fields with the most ratings of the two highest ranks, defining evaluation criteria, quantifying quality, and transparent definition of quality are the top three.

For most of the improvement potentials a few outliers exist. For this descriptive analysis, they are not excluded. For further hypotheses it may be reasonable to exclude them. The boxplot demonstrates that three mentioned fields are not evaluated with a high potential, but all of the others show nearly a high potential. In these three fields the dispersion is also the highest. In the analyses, "don't know" answers were not considered.

If you were in a position to decide on a quality improvement strategy, which three attributes would you address first?

Respondents could choose three quality attributes from a list with pre-defined quality attributes and rank them. As answers, a reduced set of quality attributes was given, consisting of: "functional suitability"; "reliability"; "performance"; "operability"; "security"; "compatibility"; "interoperability"; "maintainability"; "portability"; "installability"; and "safety". Figure 3.16 shows the relative number of answers of attributes on the first, second, and third rank.

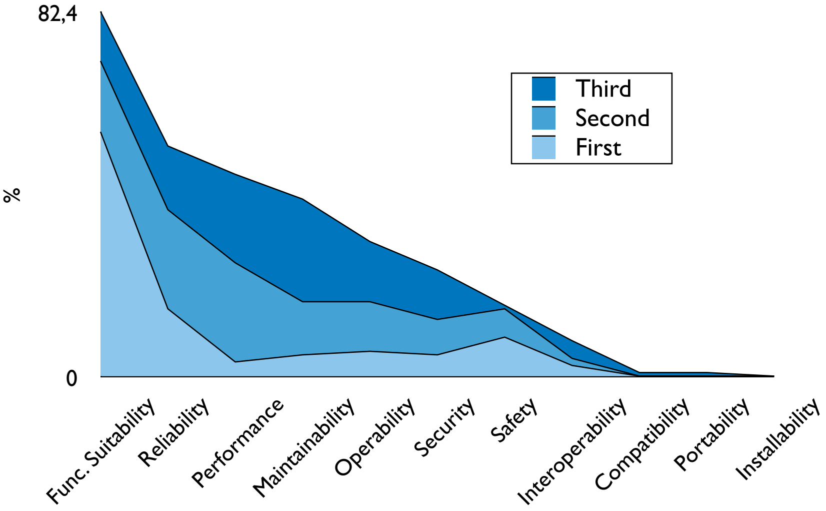


Figure 3.16: Quality Attributes to be Improved

As a result, we conclude that functional suitability, reliability, and performance are the three quality attributes ranked most often. They should be part of most quality improvement strategies. On the other hand, installability was not chosen; portability and compatibility were rarely chosen. They do not seem to be in need of urgent improvement.

3.4.7 Discussion

In general, the data shows that a wide variety of quality models are used, ranging from standards and laws to domain- and company-specific quality models. Company specific models are the most frequently applied models, being used by more than 70% of the companies. This is in line with the observation that most companies (more than 80%) adapt their quality models to their needs. We have two possible interpretations: (1) the existing quality models are unsatisfactory and thus need to be adapted (2) the adaptation of quality models is unavoidable.

The satisfaction with quality models in general is moderate. The ISO 9126 standard especially is not well accepted. It is used onyl by 28% of the respondents and three quarters of its users additionally adapted it to their needs. Thus, we conclude that existing quality models do not satisfy the needs of practice. This conclusion is highlighted by a respondent's statement in the pre-study that "the -ilities are good for management talk only" [161].

All quality attributes have been rated as important; the average importance rating is in the upper half for all quality attributes. The highest ranking were given to functional suitability, reliability, and performance; the lowest to portability. However, for almost all quality attributes there were outliers. This indicates that the importance varies for different application domains or types of software.

In line with the observation that functional suitability is considered most important, it is the quality attribute evaluated using most techniques. Testing is the predominant technique to evaluate it, followed by customer feedback and formal reviews.

Classical software testing takes a predominant role in quality assurance, as it is applied to test almost all quality attributes, except maintainability and portability which are ensured by reviews. Furthermore, testing is usually applied daily, while customer feedback and formal reviews are only applied at milestones. Code analysis and measurement are also applied daily. One explanation for the frequent use of them is their high potential for automation.

In general, the respondents saw a high potential for improvements in the field of quality models. All predefined answers were rated in the upper half of the scale. The three most important fields of improvements were the transparent definition of quality, the definition of evaluation criteria, and the quantification of quality. Both the latter hint at the importance of the development of quality models enabling the actual measurement of quality. The missing operationalization of current quality models was also highlighted by the following statement in the pre-study: "Operationalization (break down to technical attributes) is difficult" [161]

3.4.8 Threats to Validity

To ensure the validity of the constructs used in the study, we only interpreted what the question directly asked for and set it in relation to the information that we have gained from conducting the interviews of the pre-study. Moreover, we carried out a pre-test of the questionnaire, which gave additional insights for improving the design of the questions. All completed questionnaires were carefully inspected and checked for consistency where appropriate, because a Web survey is not administered and hence no additional help can be given to the participants. Respondents may have tried to manipulate the results because they have some interest in the outcome. For instance, experts in a specific quality assurance technique may have a stake in having this technique rated as very important in this study.

We assured the recipients that the responses were kept separately from the authentication tokens, so that tokens and survey responses cannot be matched. This supports that respondents report all quality-related problems located in their companies. The survey was conducted as a broad study with many different companies and individuals involved. Hence, the results should be generalizable to other companies and countries because in total the 125 participants came from 12 different countries. Most participants have long experience in software development, but experts with less experience were also among the respondents. Likewise, we cover different company sizes, different domains, and different types of systems.

3.5 The Quamoco Project

In the research project Quamoco[1] a similar research objective to that in this thesis has been pursued. It developed a quality model for software products and accompanying methods for applying the quality model in practice. The author of this thesis was deeply involved in the Quamoco project and the main contributions of both the Quamoco project and this thesis emerged during the same time-frame. Thus, there has been a back-and-forth flow of ideas between this thesis and the Quamoco project. In the following the major commonalities and differences between the results of the Quamoco project and this thesis are summarized.

- Quamoco defines an explicit meta-model for quality models [88, 158–160], which is more general and allows more degrees of freedom. The general concepts of the Quamoco model are also part of the quality model of this thesis. In this thesis, however, a more specialized meta-model is defined. In particular, it defines stricter decomposition criteria for component properties (see Section 4.4). Moreover, the meta-model in this thesis is defined formally, while in Quamoco only a prose description is given. The quality assessment approach in this thesis profoundly differs from the Quamoco approach; in Quamoco, all measurement values are weighted first and then aggregated, whereas in this thesis the aggregation and weighting steps are strictly aligned with the structure of the software product model (see Chapter 5).
- In Quamoco a "base model" [126] has been developed according to the meta-model of Quamoco. This base model is the basis for the quality model for Java in Section 6.2. The Quamoco base model was transferred to the stricter meta-model of this thesis and expanded by several additional measures. Moreover, the quality assessment part of the model was completely reworked based on the approach of this thesis. A detailed explanation of the model and its differences to the Quamoco model can be found in Section 6.2.
- The tool support built for this thesis is based on ConQAT[2] and on the tooling developed for Quamoco [27]. Due to the different meta-model and the completely different quality assessment approach the tools for this thesis are a substantial modification of the Quamoco tools.

[1] http://www.quamoco.de/, supported by the German Federal Ministry of Education and Research under grant number 01IS08023.

[2] http://www.conqat.org/

- For the case study in Chapter 8, Research Questions 2 and 3 are analogous to the case study conducted in Quamoco [89, 102, 158, 159]. Research Questions 4 and 5 are not part of the Quamoco case studies and unique to this thesis. The collection of study objects (Java systems) was initially performed for this thesis and was then partially reused for the case studies in Quamoco.

3.6 Summary

We briefly summarize the findings and insights regarding the state-of-the-art of quality models and software measurement.

The quality models with the main objective of defining quality face several problems. Their definition of quality by a taxonomical approach is criticized for several reasons: (a) the defined terms are ambiguous, incomplete, and overlapping and (b) the semantic of the tree of terms is unclear. Thus, they do not provide an applicable and concise definition of quality. Furthermore, they do not address the topic of assessing and measuring quality.

The richer successors of the hierarchical quality models address one of their shortcomings each. The Dromey's model introduced a clearer structure by the differentiation between product components and quality attributes. The activity-based quality model of Deissenboeck enhances the structure and conciseness through the modeling of activities. The SQUID model of Kitchenham introduces an explicit meta-model to achieve a clear semantic of quality models. The EMISQ model of Ploesch et al. addresses the measurement and aggregation for source code using static analysis. Even though each of these models addresses one specific topic, an integrated overall model is still missing. Furthermore, these models have limitations themselves, e.g., the activity-based model discusses only maintenance and the EMISQ model discusses only static analysis of source code.

The area of software measurement has a different focus than quality models. It focuses on the definition of software measures in general, and for source code in particular. Over time, a large number of measures have emerged. However, the meanings of the single measures for software quality are unclear. Whether and to what degree a measurement value can be used to derive a statement on the quality of a system is often unclear. Summing up, measures and their implementations in analysis tools are available today; however their meaningful application in software quality assurance remains a challenge.

The primary goal of dashboard tools is to visualize measurement values and trends. Often they also give a high-level overview of the quality of the analyzed system. Thus, they incorporate aggregation of measurement values. Since they use the hierarchical quality models as foundation for a aggregating, the dashboard tools are susceptible to the same critique as the quality models. Moreover, due to the unclear structure, the aggregated results are hardly explainable to practitioners.

The survey on quality models in practice led to the same conclusions as our review of related literature. Although quality models are widely used in practice, there is major potential for improvement. The practitioners concluded that the two major topics for improvement are the comprehensive definition of quality and operationalization.

We conclude that the three major unsolved topics with regard to quality models are the following:

- *Challenge 1:* Providing a clear definition of high-level quality attributes and clear decomposition criteria.
- *Challenge 2:* Relating the high-level quality attributes to concrete product characteristics that are measurable.
- *Challenge 3:* Providing an aggregation approach for measurement values producing comprehensible results.

4 Quality Meta-Model

In this chapter, we introduce a meta-model for quality models. In Section 4.1 we give a motivation for introducing a formal meta-model and describe the objectives we pursue with it. In Section 4.2, we provide an overview of the main concepts of our quality meta-model and describe them informally. Section 4.3 and Section 4.4 introduce the formal meta-model, subdivided into a first part defining a product model and a second part defining quality-properties and their relations. In Section 4.5 we summarize the concepts of our quality model.

4.1 Motivation & Objectives

Why a quality model?

In software engineering, quality models have been introduced to support the tasks of quality assurance (cf. Section 2.2.5). A fundamental prerequisite for supporting quality assurance is to define what *quality* means and to define concepts and terms being related to software quality. For defining abstract quality characteristics, as for instance maintainability and usability, taxonomical quality models have been introduced. To provide practical guidance in constructive quality assurance, extensive collections of guidelines and best practices have been created. For analytic quality assurance, checklists and static analysis tools are commonly used.

As we elaborated in Chapter 3, each of these approaches has its shortcomings. For instance, taxonomical quality models remain on a very abstract level, while guidelines usually miss rationales for their concrete rules. The rationales should state which quality characteristic is meant to be improved by following a concrete rule. To overcome these shortcomings, richer quality models have been proposed which integrate abstract quality characteristics and concrete properties of product components. Their objective is to capture the knowledge on quality present in taxonomical quality models, guidelines, and measurement tools in one comprehensive quality model.

Our approach for quality modeling continues this concept and seeks to describe abstract quality characteristics, concrete properties and the relations between them by one comprehensive formalism.

Why a formal quality meta-model?

According to their objective, richer quality models integrate a large variety of different information in one quality model. Hence, their primary challenge is to find an adequate mechanism for structuring information. The structuring mechanism must enable an unambiguous, non-overlapping, contradiction-free definition of terms and concepts. Furthermore, the mechanism must be able to set into relation abstract quality characteristics with concrete properties of components and with measures to quantify them. As we discussed in Chapter 3, these objectives are usually not met by these quality models. One of the prime reasons is that the semantics of these models are ambiguous and not defined precisely enough. Recent approaches for quality modeling (cf. [33, 85]) introduce informal meta-models, which improve on previous quality models, but do not solve the issue of unclear semantics sufficiently.

Our approach to overcoming this problem is to define a formal quality meta-model. We expect a formally defined meta-model to be less ambiguous than an informally defined one. This way, the interpretation of a given quality model adhering to the formal meta-model should be also clear and unambiguous. Moreover, a formal meta-model enables us to build tool support for creating quality models. The precisely-defined relations and constraints enable automatic checks for consistency and support for the quality modeler when creating and changing quality models (cf. Chapter 7 and Chapter 8).

Why include a product model?

While a formal meta-model allows us to precisely describe a structuring mechanism for quality models, defining an adequate structuring mechanism is a challenge itself. Hence, we take up and extend the idea of using a product model (cf. [33, 34]) of software as a means for structuring. The product model describes the constitution of a software product by defining the artifacts it consists of. The definition of quality characteristics and concrete properties of components then relies on the product model. Defining a product model for software is an easier task than directly defining quality characteristics and thus leads to less ambiguous results. Hence, grounding the definition of quality characteristics and properties in the product model leads to a reduction of the degrees of freedom of the quality modeler for mapping quality-related information to the quality model. We expect this to lead to a more precise and reproducible representation of quality-related information in the quality model.

The alignment of the quality characteristics and concrete properties of components with the hierarchical structure of a software product described by the product model is moreover essential for defining a quality assessment approach (cf. Chapter 5). It attaches measures to concrete properties and defines a method for aggregating measurement results. The aggregation makes use of the hierarchical structure of properties, which is in line with the hierarchical structure of the product model. This means that the strict structure of the quality model enables us to define a quality assessment approach, which fully relies on the quality model.

4.2 Overview

In Section 2.1, we discussed how reasoning about quality essentially means defining the properties of things. Typical properties we are reasoning about in quality models are, for instance, *maintainability* of software products, *complexity* of source code, or *conciseness* of identifiers in source code. Thus, the central task of a quality meta-model is to provide concepts for defining and organizing properties.

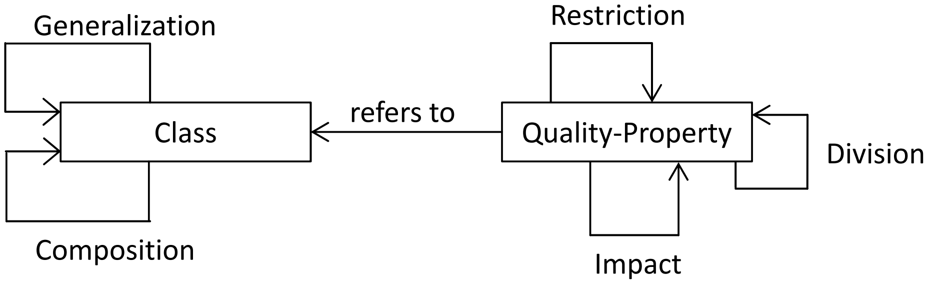


Figure 4.1: Main Concepts of the Meta-Model Represented as Ontology

For organizing properties in a structured way, we use a product model of software as the backbone, as proposed by several quality models in literature [33, 34]. In contrast to these quality models, we use the object-oriented paradigm as realized in UML class diagrams [116, 132] for defining the product model.

Figure 4.1 shows the main concepts and relations of our quality meta-model as an ontology. The product model is defined via the concept *Class* and the relations *Generalization* and *Composition*. The prime element for describing quality-related concepts is *Quality-Property*, whereby the definition of a quality-property is always based on a class of the product model. Between quality-properties there are three relations, *Restriction*, *Division*, and *Impact*. In the following, the concepts and relations are defined and illustrated by examples.

Class

Definition. We define a class as a collection of objects exhibiting some common characteristics, so that they are distinctly identifiable (cf. [132, p. 42]). Since our quality model targets product quality, the classes of the quality model represent artifacts and ideas of software products.

Purpose. The main purpose of classes is to provide a basis for defining properties. It is much easier to first define a product model in the form of classes and then to derive a property model, than to directly define a property model. Furthermore, the hierarchical constitution of the software product represented in the product model is needed for aggregating measurement results when conducting quality assessments.

Example. Figure 4.2 shows an excerpt of the quality model for Java source code presented in detail in Section 6.2. The classes in this example describe parts of source code in the programming language Java. The class *Product* is the root and represents the software product. The product consists of *Source code construct*s, which include *Java Class*, *Method*, *Field*, and *Expression*. Entirely different concepts can also be described by classes; for instance, in a quality model describing user interfaces, the classes could be *windows*, *buttons*, *text fields*, and *lists*.

Generalization

Definition. We define a *generalization* relationship between a more general and a more specific class (cf. [132, p. 51]). The more specific class comprises objects with some special characteristics distinguishing them from the objects of the more general class. Nonetheless, the special class is

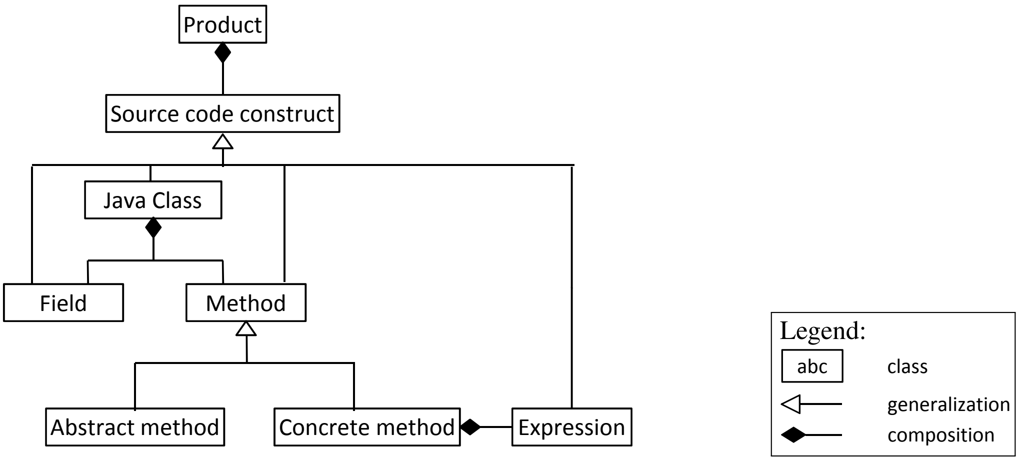


Figure 4.2: Example of Classes and their Relations
(Excerpt from the quality model for Java source code presented in Chapter 8)

fully consistent with the general class, i.e., all objects of the specific class are also contained in the general class. The general class is called a *generalization* of the specific class. Conversely, the specific class is called a *specialization* of the general class.

Purpose. The purpose of the generalization relationship is to abstract from specific classes to more general ones. This way, common quality-properties of several specific classes can be defined once for the more abstract class, and they are inherited by the more specific classes. This prevents the redundant definition of quality-properties for different classes. Nonetheless, differences between the special classes can be explicitly modeled. Hence, the generalization relation "permits the incremental description of an element" [132, p. 52] by starting with a general class and continuing with more special classes.

Example. In the example in Figure 4.2, the generalization relation is used to describe that the classes *Field*, *Java Class*, *Method*, and *Expression* are a specialization of *Source code construct*. Furthermore, *Abstract method* and *Concrete method* are a specialization of *Method*.

Composition

Definition. A composition represents a container-part relationship between two classes. It describes how a class, called a *container*, consists of other classes, called *parts* (cf. [132, p. 49]).

Purpose. The purpose of this relationship is to describe the hierarchical constitution of a software product. This view onto classes is a natural one for people thinking about software products. Thus, this relation is convenient for presenting the product model to the quality modeler. When conducting quality assessments, such a description of the hierarchical constitution of a software product is important too. It is used for aggregating measurement results in order to get to a quality statement for the entire software product.

Example. In the example in Figure 4.2, the composition relation is used to describe that a *Java Class* contains *Fields* and *Methods*. It is also used to describe that a *Concrete method* contains *Expressions*.

Quality-Property

In Section 2.1.1, we thoroughly discussed the notion of *properties* and concluded that in general properties describe *attributes*, *qualities*, *features*, or *characteristics* of a thing. In our quality model, we use a narrower definition of properties, called *quality-properties*.

Definition. Since in the quality model the software product is modeled in the form of classes, *a quality-property always refers to a class* it characterizes. Moreover, all quality-properties are defined in a *quantitative manner*. This means that a quality-property assigns to each object of its class a value expressing the degree to which the object manifests the quality-property. This degree is expressed on a continuous scale between 0 and 1. The value of 0 means that the object does not manifest the quality-property; we can also say the quality-property is *not satisfied* by the object. The value of 1 means the object fully manifests the quality-property, in which case we say that the quality-property is *satisfied* by the object.

Purpose. Quality-properties are the main construct in our quality model for describing quality-related concepts. They are used for both abstract concepts like quality attributes and concrete technical concepts like source code-based measures. In this way, the concept of quality-properties in the quality model fulfills the purpose of describing diverse quality-related concepts in a consistent manner.

Example. An example of a rather abstract quality-property is *Maintainability* of a *Product*, describing whether the software product can be effectively and efficiently maintained. In this case, *Product* is the class to which the quality-property refers. An example of a concrete technical quality-property is the *Detail complexity* of a *Method*, stating that a method in source code "is complex because it consists of many parts" [126].

Restriction

Definition. A restriction is a relationship between a general and a specific quality-property. It is the analogy of the specialization (inverse of generalization) relationship on classes. Hence, the class of the specific quality-property must be in a generalization relation to the class of the general quality-property. Moreover, the specific quality-property must be consistent with the general quality-property, i.e., it must yield the same degrees of satisfaction.

Purpose. The purpose of this relationship is to enable the definition of general quality-properties that are based on general classes and to inherit these general quality-properties to more special classes. This way, a general quality-property is broken down to more specific quality-properties, which can be described more precisely. At the same time, the commonalities between several more specific quality-properties are explicitly captured by their relation to the more general quality-property.

When creating a quality model, this relationship enables a top-down procedure. First, general quality-properties are defined for general classes. Then, step by step, the general quality-properties are restricted by more specific quality-properties, whereby the generalization relationship of classes prescribes which restrictions are possible. This way, guidance for choosing restrictions is given to the quality modeler.

Example. Figure 4.3 shows an excerpt of the quality model for Java source code presented in Chapter 8. The quality-property *Detail complexity* of *Source code construct* describes that there is a

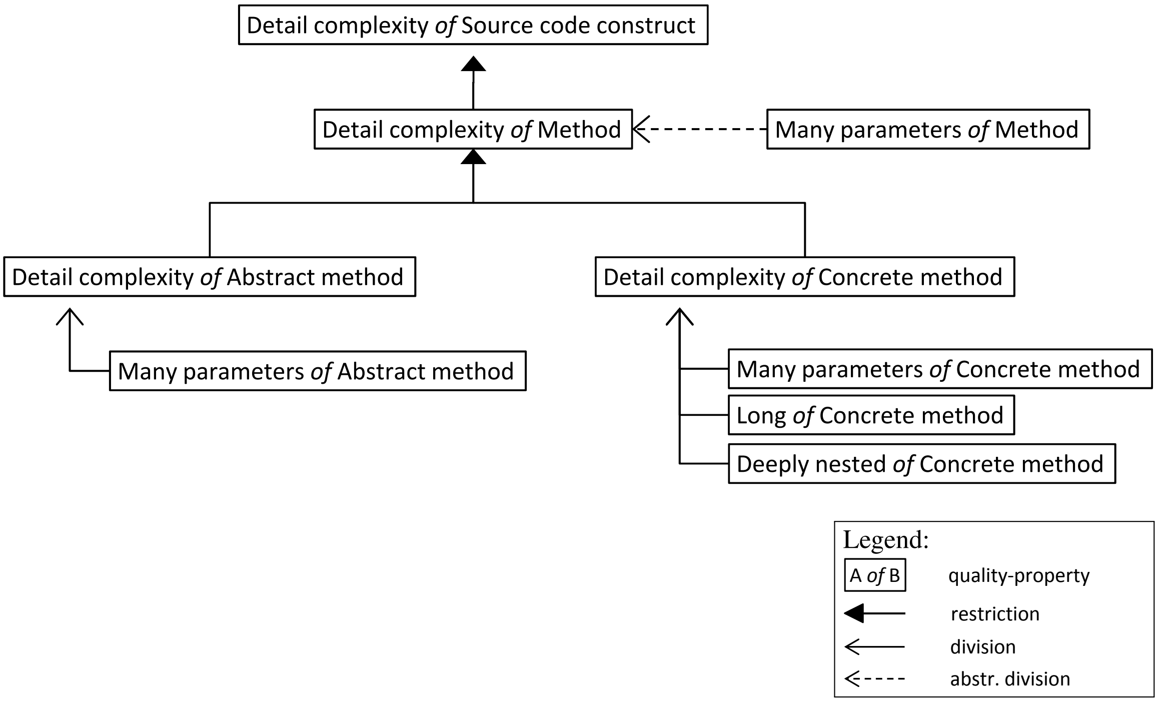

Detail complexity *of* Source code construct
Detail complexity *of* Method
Many parameters *of* Method
Detail complexity *of* Abstract method
Detail complexity *of* Concrete method
Many parameters *of* Abstract method
Many parameters *of* Concrete method
Long *of* Concrete method
Deeply nested *of* Concrete method
Legend:
A *of* B
quality-property
restriction
division
abstr. division

Figure 4.3: Example of Quality-Properties, Restriction, and Division Relations
(Excerpt from the quality model for Java source code presented in Chapter 8)

general notion of detail complexity for source code constructs. According to the specialization of the class *Source code construct* to *Method*, which is specialized to *Abstract* and *Concrete method*, the restriction takes place. This results, e.g., in the quality-property *Detail complexity* of *Concrete method*, which can be defined more precisely. This model also captures the commonality between the *Detail complexity* of *Abstract methods* and the *Detail complexity* of *Concrete methods*, by explicitly modeling that both are restrictions of the *Detail complexity* of *Methods*.

Division

Definition. A division is a breakdown relationship between two quality-properties. A quality-property that is divided into several other quality-properties can be expressed as a combination of those quality-properties. Hence, two quality-properties being in a division relation always refer to the same class.

Purpose. The purpose of this relationship is to break down a complex quality-property into more concrete, tangible ones. While the restriction relationship approaches to more concrete quality-properties by specializing the class they refer to, this relationship stays on the same granularity of classes. This way, it represents an orthogonal way of breaking down more general quality-properties into more concrete ones.

Example. In Figure 4.3, the division relation was used to describe that the quality-property *Detail complexity* of *Concrete method* can be broken down into *Many parameters*, *Long*, and *Deeply-nested*.

Abstract Division and Inheritance

Definition. An abstract division is a variant of the division relationship, where the quality-property which is divided is not expressible by the combination of the other quality-properties. Thus, this relation is called *abstract* and its only purpose is to be inherited alongside the restriction relation. This means, an abstract division defined for a more general quality-property is inherited by the more specific quality-properties that are in a restriction relation to the more general quality-property.

Purpose. The inheritance of the abstract division relation enables us to define an abstract division once for the general quality-property and for it to be inherited by the more specific quality-properties. Hence, this construct prevents the redundant definition of divisions for specific quality-properties, by transferring it to the more general quality-property and then it being inherited.
The inheritance also serves the purpose of extending the top-down approach for constructing a quality model to the division relation. First, a quality modeler adds an abstract division to a general quality-property. Then, he adds more specific quality-properties to the general quality-properties by a restriction relation; the dividing quality-properties of the general quality-property are inherited by the newly defined specific quality-properties, for which they can be defined more precisely.

Example. Figure 4.3 shows an example for an abstract division and its inheritance. By an abstract division we define that *Detail complexity* of *Methods* can be broken down into *Many parameters* of *Methods*, which describes that a method has a large number of parameters.
Since *Detail complexity* of *Method* is restricted by two quality-properties for the classes *Abstract method* and *Concrete method*, the inheritance of abstract divisions comes into effect: The quality-property *Many parameters* is inherited by these quality-properties.
For *Concrete methods* the quality-property *Detail complexity* can be further divided into the quality-property *Long*, meaning that a method consists of a large number of statements and *Deeply nested* meaning that the nesting depth of the method is high.

Impact

Definition. An impact is a relationship between a source and target quality-property, whose classes are in a composition relation and where there is a certain type of dependency between the two quality-properties. We distinguish between two types of dependency: A *positive* dependency which means that the value of the target quality-property increases if the value of the source quality-property increases; a *negative* dependency which means that the value of the target quality-property decreases if the value of the source quality-property increases.

Purpose. The purpose of this relationship is to describe how specific (often technical) quality-properties influence general quality-properties, as, for instance, quality attributes. This way, it is possible to describe the relation of detailed technical quality-properties and general quality attributes in a consistent manner.
Describing this kind of influence between different quality-properties is essential for concretizing abstract quality characteristics. Abstract quality characteristics usually describe attributes of the entire product and thus refer to the class *Product*, as, for instance, *Maintainability* of *Product*. The main goal of quality models, which is to break down such quality characteristics to tangible properties, cannot be achieved exclusively by restriction or division. Doing so would just resemble the taxonomical/hierarchical quality models described in Section 3.1, which do not achieve this goal.

Thus, explicitly capturing the dependencies between concrete quality-properties and abstract quality attributes is the only way to get a connection between abstract and concrete quality-properties.

Example. For instance, we define that the quality-property *Detail complexity* of *Method* has a negative impact on the quality-property *Maintainability* of *Product*, because from empirical investigations (e.g., [23, 92]) we know that complex methods demand more effort to be understood and thus decrease the maintainability of the software product.

4.3 Objects and Classes

In Section 4.2, we informally described a data model in the style of UML class diagrams and object-oriented programming languages. In this section, we introduce a formal definition of this model. In literature, several approaches for formalizing the UML are known. Early approaches describe how a UML model can be transformed into a formal specification, for instance Z [38]. Later approaches discuss the meaning of syntax and semantics in the context of UML and provide formal models for UML [16, 17, 56, 98, 128]. These approaches formalize large parts of the UML, including behavioral characteristics represented by state machines or message sequence charts. Regarding class diagrams, they formalize classes including attributes, associations, and multiplicities of associations. In this thesis, just a fraction of these constructs is needed. Therefore, we provide our own formalization of the constructs *class*, *object*, *generalization*, and *composition*. Our formalization approach is, however, based on the main considerations of those in the literature. For instance, we define classes as sets of objects, as do all other approaches in the literature. By introducing our own formalization, we achieve tailored definitions for our purpose and a consistent terminology and style throughout this entire chapter.

Definition 4.1 Universe of Discourse. *The set of all objects that are relevant when talking about software products is the universe of discourse:*

$$\mathrm{U}$$

As defined in Section 4.2, a class denotes a set of objects with some common characteristics so that they are distinctly identifiable. The set of all possible classes is $\wp(\mathrm{U})$. We define a subset of it as all classes we are talking about in the quality model.

Definition 4.2 Set of all Classes. *The set of classes* C *in the quality model is a finite subset of all possible classes* $\wp(\mathrm{U})$*:*

$$\mathrm{C} \subseteq \wp(\mathrm{U})$$

Furthermore, we exclude the existence of an empty class.

Axiom 4.1 No Empty Class.

$$\varnothing \notin \mathrm{C}$$

In a quality model, we want to ensure all objects are described by classes. Thus, we require that each object is a member of a class. All objects contained in a class are called *instances* of it.

Axiom 4.2 Members of Classes. *Each object of the universe of discourse is a member of a class:*

$$\forall o \in \mathrm{U}: \ \exists c \in \mathrm{C}: \ o \in c$$

4.3.1 Generalization

The generalization relation (see Section 4.2), describes a relationship between a more general and a more specific class. The more specific class contains a subset of the objects of the more general class. The objects of the more specific class have some special characteristic distinguishing them from the objects of the more general class.

We define a class as a generalization of another class, if and only if it is a superset of it. Accordingly, a class is a specialization of another class, if it is a subset of it.

Definition 4.3 Generalization Relation. *A class* $c_1 \in \mathrm{C}$ *is a generalization of a class* $c_2 \in \mathrm{C}$ *if and only if* $c_2 \subset c_1$.

If $c_1 \subset c_2$ then c_1 is called a *subclass* of c_2 and c_2 is called a *superclass* of c_1. Since the generalization relation directly resembles the subset relation of sets it is acyclic and transitive, as expressed in the following theorem.

Theorem 4.1 Generalization Relation is Acyclic and Transitive.

Proof. *This directly follows from Definition 4.3.*

As explained in Section 4.2, in our quality model we want to use the class model for structuring quality-properties, which will be defined on classes. Thus, the graph spanned by the generalization relation should be tree-like. To achieve this, we decide that there must be no overlapping between classes, meaning that no object is a member of more than one class, unless the classes are in a generalization relation.

Axiom 4.3 No Overlapping Classes. *Two classes are either in a generalization relation or they have no joint elements:*

$$\forall c_1, c_2 \in \mathrm{C}: \;\; (c_1 \neq c_2) \;\; \Rightarrow \;\; ((c_1 \subset c_2) \vee (c_2 \subset c_1) \vee (c_1 \cap c_2 = \varnothing))$$

This axiom constrains the structure of the classes in a way that two typical characteristics of object-oriented data models are achieved. These two characteristics are formulated as two theorems in the following: First, there is *no multiple instantiation*, meaning that each object is instance of exactly one class, not considering its transitive superclasses. Second, there is *no multiple inheritance*, meaning that each class has exactly one superclass, not considering the transitive superclasses.

Theorem 4.2 No Multiple Instantiation. *For an object o, there do not exist two distinct classes* c_1 *and* c_2 *so that o is instance of both of them and they are not in a generalization relation.*

$$\forall o \in \mathrm{U}: \; \nexists\, c_1, c_2 \in \mathrm{C}: \;\; (c_1 \neq c_2) \wedge (c_1 \not\subset c_2) \wedge (c_2 \not\subset c_1) \wedge (o \in c_1) \wedge (o \in c_2)$$

Proof. *This directly follows from Axiom 4.2 and Axiom 4.3.*

Theorem 4.3 No Multiple Inheritance. *For a class c, there do not exist two classes c_1 and c_2 so that they are not in a generalization relation and c is a subclass of both of them.*

$$\forall c \in \mathrm{C}: \nexists\, c_1, c_2 \in \mathrm{C}: \quad (c_1 \neq c_2) \wedge (c_1 \not\subset c_2) \wedge (c_2 \not\subset c_1) \wedge (c \subset c_1) \wedge (c \subset c_2)$$

Proof. *This directly follows from Axiom 4.2 and Axiom 4.3.*

Since the set of classes is finite (Definition 4.2) and there is no multiple instantiation (Theorem 4.2), for each object we can define a unique class, called the *class of the object*, which is the smallest class of which the object is an instance.

Definition 4.4 Class of an Object. *The* class of an object *is a function* class : U → C*, mapping each object to the smallest class it is an instance of:*

$$\mathrm{class}(o) \quad \mapsto \quad min_{\subset}(\{c \in \mathrm{C}: \ o \in c\})$$

From Axiom 4.2, it follows that at least one such class exists. From the finiteness of C *and Axiom 4.3, it follows that there is at most one such class. Hence, there is always exactly one such class.*

4.3.2 Composition

According to the definition in Section 4.2, the composition relation is used to describe the hierarchical constitution of a software product. Thus, it describes which objects consist of other objects.

Definition 4.5 Composition Relation on Objects. *The composition relation* COMP_U *describes a composition of objects.*

$$\mathrm{COMP}_\mathrm{U} \ \subseteq \ \mathrm{U} \times \mathrm{U}$$

For predefining how a software product is represented in the form of objects, we define a composition relation of classes. This relation describes which classes are composed of other classes; i.e., it describes that objects of a class may be composed of objects of another class.

Definition 4.6 Composition Relation on Classes. *The composition relation describes which classes are composed of other classes:*

$$\mathrm{COMP} \ \subseteq \ \mathrm{C} \times \mathrm{C}$$

As stated before, the composition of objects must adhere to the constraints defined by the composition on classes. Thus, we introduce the following axiom.

Axiom 4.4 Consistency of Composition on Objects and Classes. *If two objects are in a composition relation, then their classes are in a composition relation too.*

$$\mathrm{COMP}_\mathrm{U}(o_1, o_2) \ \Rightarrow \ (\mathrm{class}(o_1), \mathrm{class}(o_2)) \in \mathrm{COMP}$$

This means that two objects may only be in a composition relation if their classes are in a composition relation. This way, the composition relation of classes describes which compositions on the object level are allowed. On the other hand, if two classes are in a composition relation, it does not mean that all their objects have to be in a composition relation.

For two classes $(c_1, c_2) \in \mathrm{COMP}$ we call c_1 the container of c_2 and c_2 a part of c_1.

The composition relation between classes is an essential part of the class model. Since the class model is used as a main structuring mechanism in the quality model, it should be easily understandable and manageable. On the other side, the hierarchical structure of the software product will be used in quality assessments for aggregating measurement values. Because the goal is to come to an overall quality assessment for an entire software product, we require the class hierarchy to be a tree with a unique root class. A tree also satisfies the requirement of being easily understandable and manageable. Since the tree of classes represents the software product, the root-class is called $\mathrm{product} \in \mathrm{C}$.

Axiom 4.5 Composition Relation COMP is a Tree.

Definition 4.7 Root Class $\mathrm{product}$. *The root class of the composition relation is called* $\mathrm{product} \in \mathrm{C}$.

In the following considerations, we will often reason about the transitive closure of the composition relation. Thus, we explicitly introduce it as COMP^*.

Definition 4.8 Transitive Closure of the Composition Relation. *The transitive closure of the composition relation is denoted as* COMP^*.

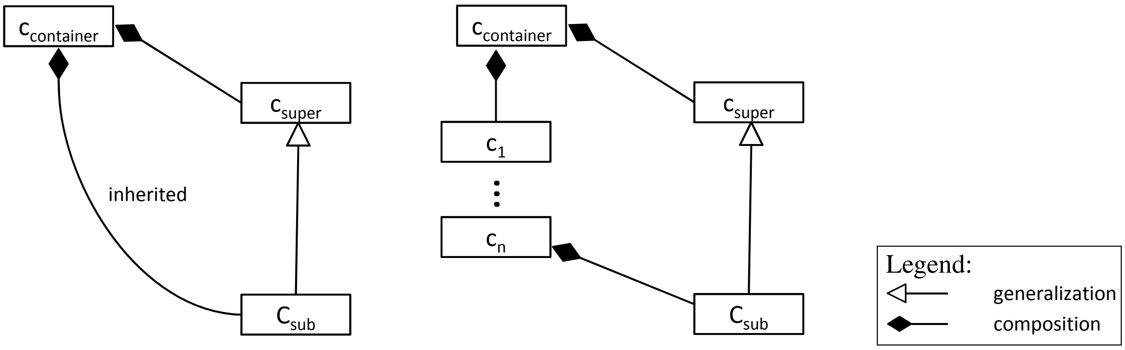
$$\mathrm{COMP}^* = \bigcup_{i \in \mathbb{N}_{>0}} \mathrm{COMP}^i$$

whereby COMP^i *denotes the* i*-times applied composition of relations.*

4.3.3 Inheritance of the Composition Relation

When creating a product model of software products using the generalization and composition relation on classes, both these relations are used together. For achieving an acyclic and contradiction-free structure of the combined graph spanned on classes, we define a constraint on the usage of the composition relation: We call it the *inheritance* of the composition relation alongside the generalization relation. It means that a subclass of a superclass gets the same container as the superclass by default. The default container can then be overwritten by parts of it. The thought behind this definition is that a subclass cannot have a more general container then a superclass. For instance, if identifiers are contained within methods, then more special types of identifiers can only be contained within methods or parts of methods.

By enforcing an acyclic and contradiction-free class model, the inheritance of the composition relation leads to an easily understandable class model. Furthermore, this constraint enables active



c_{sub} is a direct part of $c_{container}$.
(a) Simple Case

c_{sub} is transitively a part of $c_{container}$.
(b) With Overwriting

Figure 4.4: Inheritance of the Composition Relation

support of the quality modeler by tools, because when introducing a new sub-class, a tool can propose to use the inherited container by default.

Figure 4.4a visualizes the concept of inheritance. In the simple case, a class c_{sub} has the superclass c_{super}, which has the container $c_{container}$. According to the inheritance, $c_{container}$ is also the default container of c_{sub}. In Figure 4.4b we show the case with overwriting. Instead of using $c_{container}$ as container of c_{sub}, a part of $c_{container}$ is used as container of c_{sub}.

Axiom 4.6 Inheritance of Composition. *Given a class c_{sub}, which is a subclass of c_{super}, a class $c_{container}$ which is the container of c_{super}, then c_{sub} is a part of $c_{container}$.*

$$\begin{aligned}&\forall c_{sub}, c_{super}, c_{container} \in \mathrm{C}:\\ &\quad (c_{sub} \subset c_{super}) \wedge ((c_{container}, c_{super}) \in \mathrm{COMP}) \;\Rightarrow\; (c_{container}, c_{sub}) \in \mathrm{COMP}^*\end{aligned}$$

4.4 Quality-Properties

According to our discussion in Section 2.1, properties in general are interpreted as predicates, which hold for all objects manifesting a certain property [145]. As we discussed in Section 4.2, in our quality model, we are modeling a more specific type of property, called quality-property. Quality-properties are a portion of all general properties, having two special characteristics: (1) they have a class of the class model as domain and (2) they are quantitative properties, meaning that they describe a gradual possession of a property by an object. For expressing the gradual possession we use the Łukasiewicz fuzzy propositional calculus (according to Hajek et al. [53, p. 63] and Bohme et al. [15, p. 209]). Summarized in brief, instead of a two-valued logic with truth values *true* and *false*, the truth value in fuzzy logic is between 0 and 1. In the quality model 1 means the object fully satisfies the quality-property and 0 means the object does not satisfy the quality-property. All values in between are degrees of possession.

Definition 4.9 Quality-Property. P *is the set of all quality-properties that are defined in the quality model. A quality-property* $p \in \mathrm{P}$ *is a function mapping objects of a certain class* $c \in \mathrm{C}$ *to a logical value:*

$$p: \ c \to [0,1]$$

As we have seen in Section 4.2, in the quality model it is important to have an informal description of quality-properties that is easily understandable. Often we want to use the same name for different quality-properties and distinguish them only by their domain. Thus, we introduce the following notation:

Definition 4.10 Notation for Quality-Properties. *For a quality-property* $p \in \mathrm{P}$ *we use the following notation:*

$[p|\operatorname{dom}(p)]$, *pronounced as "$p$ of* $\operatorname{dom}(p)$*".*

4.4.1 Restriction

As explained in Section 4.2, a restriction describes a relationship between a more general and a more specific quality-property. Since it is analogous to the specialization relationship (inverse of generalization) on classes, the domain of the more specific quality-property must be a subset of the domain of the more general quality-property.

Definition 4.11 Restriction of Quality-Properties. *The relation* $\mathrm{RES} \subseteq \mathrm{P} \times \mathrm{P}$ *describes which quality-properties are restricted by other quality-properties. A quality-property* p *is restricted by a quality-property* p_i*, if and only if* p_i *is a projection of* p*.*

$$(p_1, p_2) \in \mathrm{RES} \quad \Leftrightarrow \quad (\operatorname{dom}(p_2) \subset \operatorname{dom}(p_1)) \ \wedge \ (\forall o \in \operatorname{dom}(p_2): \ p_1(o) = p_2(o))$$

$(p_1, p_2) \in \mathrm{RES}$ means that p_1 is restricted by p_2. It is obvious that the restriction relation is closely aligned with the generalization relation between classes, leading to the following theorem.

Theorem 4.4 Restriction Relation is Transitive and Acyclic.

Proof. *This follows from the definition of the restriction relation (Definition 4.11) and the transitivity and acyclicity of the generalization relation (Theorem 4.1).*

The purpose of this relationship is to enable the definition of general quality-properties based on general classes, which are then restricted to more special classes. The resulting, more specialized quality-properties can be described more precisely. The informal usage of this relationship in the quality model is to create a taxonomy of quality-properties. Thus, we require this relation to be tree-like.

Axiom 4.7 Restriction is Tree-Like. *In the graph spanned on quality-properties by the restriction relation, each connected component is a tree, not considering transitive restrictions.*

If a quality-property p is restricted by quality-properties $p_1,\ p_2,\ \ldots,\ p_n$, then the restricting quality-properties cover distinct parts of the domain of the quality-property p (if the transitive restrictions are not considered). It is, however, not assured that all the domain of the restricted quality-property is covered by the restricting quality-properties. If the domain is fully covered by the restricting quality-properties, we call the restricted quality-property *complete regarding restriction*.

Definition 4.12 Completeness of Restriction. *A quality-property* $p \in \mathrm{P}$ *is called complete regarding restriction if it is restricted by quality-properties* $p_1, \ldots, p_n$ *and if its domain is fully covered by its restricting quality-properties:*

$$\mathrm{complete}_{\mathrm{RES}}(p) \quad \Leftrightarrow \quad \mathrm{dom}(p) = \bigcup_{\{p_i:\ (p,p_i)\in\mathrm{RES}\}} \mathrm{dom}(p_i)$$

The characteristic of being complete is important for the quality assessment approach (cf. Chapter 5), because the hierarchy of quality-properties created by the restriction relation is used to aggregate measurement values. The aggregation is only meaningful if all the domain of a quality-property is covered by the restricting properties, i.e., an aggregation is only meaningful for complete restrictions.

4.4.2 Division

In Section 4.2, we explained that the division relationship describes a breakdown between quality-properties. If a quality-property is divided into several other quality-properties, then it is expressible by a combination of the dividing quality-properties. As a consequence, the domain of a quality-property and its dividing quality-properties is always the same.

Usually, there are multiple ways of dividing a quality-property into other quality-properties. Thus, we explicitly introduce a semantical division relation, $\mathrm{DIV}_{\mathrm{all}}$ to capture all possibilities for dividing a quality-property. Based on this semantical relation, we define a syntactical relation, DIV, describing the actual divisions used in one quality model. The divisions used in this relation are selected out of the $\mathrm{DIV}_{\mathrm{all}}$ relation by the quality modeler during the process of developing the quality model.

Definition 4.13 Possibilities for Dividing a Quality-Property. *The relation* $\mathrm{DIV}_{\mathrm{all}} \subseteq \mathrm{P} \times \wp(\mathrm{P})$ *describes for each quality-property* p*, the set of quality-properties* D *into which it is dividable. A quality-property* p *is dividable into a set of quality-properties* D*, written as* $(p, D) \in \mathrm{DIV}_{\mathrm{all}}$*, if and only if:*

1. *The quality-property* p *is not divided into the empty set:*

   $D \neq \emptyset$

2. *The quality-property* p *is not divided into itself:*

   $p \notin D$

3. *The quality-property* p *has the same domain as all quality-properties of* D*:*

   $\forall p_i \in D: \ \mathrm{dom}(p) = \mathrm{dom}(p_i)$

4. *The quality-property* p *is calculable by a function* $f : [0,1]^n \to [0,1]$ *using the values of the dividing quality-properties:*

   $\exists f: \ \forall o \in \mathrm{dom}(p): \ p(o) = f(p_1(o), \ldots, p_n(o))$ *, whereby* $D = \{p_1, \ldots, p_n\}$

5. *The set* D *is minimal, i.e., if an element of* D *is removed, then at least one of the conditions 1–4 is no longer satisfied.*

Definition 4.14 Division Relation. *The relation* $\mathrm{DIV} \subseteq \mathrm{P} \times \mathrm{P}$ *describes for each quality-property* p*, the set of quality-properties into which it is divided. This relation is consistent with the* $\mathrm{DIV}_{\mathrm{all}}$ *relation, i.e., the set of dividing quality-properties is contained in the* $\mathrm{DIV}_{\mathrm{all}}$ *relation:*

$$\forall p \in \mathrm{P}: \ \forall D \subseteq \mathrm{P}: \ \left(\begin{array}{lll} & \forall p_i \in D: & (p, p_i) \in \mathrm{DIV} \\ \wedge & \forall p_i \in \mathrm{P} \setminus D: & (p, p_i) \notin \mathrm{DIV} \end{array} \right) \ \Rightarrow \ (p, D) \in \mathrm{DIV}_{\mathrm{all}}$$

The $\mathrm{DIV}_{\mathrm{all}}$ relation describes all possibilities for breaking down a quality-property p into a set of other quality-properties. This includes two common cases: (1) a quality-property p can be broken down in multiple different ways. For instance, p can be broken down into $\{p_1, p_2\}$ or into $\{p_3, p_4\}$. (2) The breakdown of quality-properties can be cyclic. For example p_1 is broken down into $\{p_2, p_3\}$ and p_2 can be broken down into $\{p_1, p_3\}$. In a quality model, however, we want to use the division relation in a taxonomical way for a hierarchical breakdown of quality-properties. Thus, regarding the first case, the quality modeler has to choose one possibility of breakdown for the actual quality model. This is guaranteed by the definition of the relation DIV. Regarding the second case of cyclicity, we require the division relation to be tree-like. We state this explicitly in the following axiom.

Axiom 4.8 Division Relation DIV is Tree-like. *In the graph spanned on quality-properties by the division relation* DIV*, each connected component is a tree.*

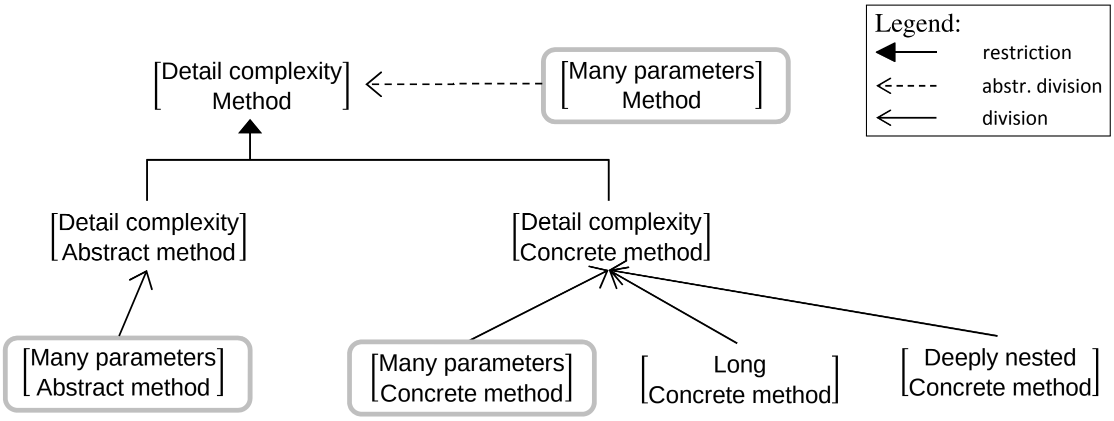
Summing up, the division relation DIV represents a taxonomical breakdown of quality-properties. A quality-property which is broken down into other quality-properties by a division is always calculable using the dividing quality-properties.

4.4.3 Abstract Division Relation and its Inheritance

In Section 4.2, we explained that we introduce an abstract division relation and an inheritance of this relation alongside the restriction relation. This serves the purpose of explicitly modeling commonalities between two different divisions.

Figure 4.5, shows an example. For instance, the quality-property [*Detail complexity*|*Concrete method*] is divided into [*Many parameters*|*Concrete method*]. The quality-property [*Detail complexity*|*Abstract method*] is also divided into [*Many parameters*|*Abstract method*]. Both the quality-attributes called *Many parameters* are restrictions of a general quality-attribute [*Many parameters*|*Method*] (the figure does not show this restriction relation for the sake of a clear visualization). In this case, an *abstract division* of [*Detail complexity*|*Method*] into [*Many parameters*|*Method*] is introduced. This way, it is explicitly captured that a division into *Many parameters* is possible for *Method*s. For [*Detail complexity*|*Method*], this relation is an *abstract* division and not a division according to Definition 4.14, because its value cannot be calculated based on its dividing quality-properties. For the more concrete quality-properties referring to *Abstract method* and *Concrete method*, however, it is a division according to Definition 4.14, because their values can be calculated based on the dividing quality-properties.

This relation serves two purposes: First, the clarity and understandability of the model is increased, because the commonalities between two different divisions are explicitly modeled in the quality model. Second, in a top-down approach for creating the quality model, the quality modeler can first define the abstract division and then it is guaranteed that the modeler does not forget to introduce the corresponding dividing quality-properties.


Legend:
restriction
abstr. division
division
Detail complexity Method
Many parameters Method
Detail complexity Abstract method
Detail complexity Concrete method
Many parameters Abstract method
Many parameters Concrete method
Long Concrete method
Deeply nested Concrete method

For the clarity of the visualization, we omitted the restriction relations between [*Many parameters*|*Method*] and both [*Many parameters*|*Abstract method*] and [*Many parameters*|*Concrete Method*]. They are marked by rounded rectangles.

Figure 4.5: Example: Inheritance of Divided Quality-Properties

We define the abstract division relation $\mathrm{DIV}_{\mathrm{abs}}$ formally as follows.

Definition 4.15 Abstract Division Relation. *The abstract division relation* $\mathrm{DIV}_{\mathrm{abs}} \subseteq \mathrm{P} \times \mathrm{P}$ *defines which quality properties are in an abstract division relation. A quality-property p_a is an abstract division of a quality-property p, iff there is at least one restricting quality-property of p and iff for each restricting quality-property p_r of p there is a quality-property p_d, which is in a division relation to p_r and which is a restriction of p_a.*

$$\begin{tikzcd} p & p_a \arrow[l, dashed] \\ p_r \arrow[u] & p_d \arrow[l] \arrow[u] \end{tikzcd}$$

$$\begin{aligned}(p, p_a) \in \mathrm{DIV}_{\mathrm{abs}} \quad \Leftrightarrow \quad & \exists p_r \in \mathrm{P} : \ (p, p_r) \in \mathrm{RES} \quad \wedge \\ & \forall (p, p_r) \in \mathrm{RES} : \ \exists p_d \in \mathrm{P} : \ (p_r, p_d) \in \mathrm{DIV} \ \wedge \ (p_a, p_d) \in \mathrm{RES}\end{aligned}$$

In the following, we explain the top-down approach to creating a quality model using the abstract division relation. In the quality model, the quality-property [*Detail complexity*|*Method*] is defined. The quality modeler knows that the detail complexity of methods has in general to do with its number of parameters. Thus, the modeler captures this relation by modeling that [*Detail complexity*|*Method*] is abstractly divided into [*Many parameters*|*Method*]. Since [*Detail complexity*|*Method*] cannot be calculated based on [*Many parameters*|*Method*], it cannot be modeled as a division relation according to Definition 4.14.

Next, the quality modeler restricts the quality-property [*Detail complexity*|*Method*] to a more specific quality-property [*Detail complexity*|*Concrete method*]. The inheritance of the abstract division by the concrete quality-property prescribes that [*Detail complexity*|*Concrete method*] is divided into [*Many parameters*|*Concrete method*]. Together with other dividing properties, like *Long* and *Deeply nested*, the divided quality-property is calculable based on the dividing quality-properties and thus in the division relation according to Definition 4.14.

If the quality modeler then introduces a second quality-property which restricts [*Detail complexity*|*Method*], as, for instance, [*Detail complexity*|*Abstract method*], then the inheritance assures that a dividing quality-property [*Many parameters*|*Abstract method*] is also created.

4.4.4 Impact

In Section 4.2, we defined that an impact is a relationship between a source and target quality-property, whose classes are in a composition relation and where there is a dependency between the two quality-properties. A positive dependency means that the value of the impacted quality-property increases if the value of an impacting quality-property increases, while a negative dependency means that the value of the impacted quality-property decreases if the value of an impacting quality-property increases. We call an impact with a positive dependency a *positive impact* and an impact with a negative dependency a *negative impact*.

A semantical definition of an impact according to this informal description usually leads to multiple possibilities for defining impacting quality-properties for one quality-property. If, for instance, a quality-property has a positive impact on another quality-property, then its negation has a negative impact. Thus, we define a semantical relation, $\mathrm{IMP}_{\mathrm{all}}$ describing all possibilities for defining impacts for one quality-property. Based on this semantical relation, we define a syntactical relation

IMP, describing the actual impacts in one quality model. The impacts used in this relation are selected out of the $\mathrm{IMP}_{\mathrm{all}}$ relation by the quality modeler during the process of developing the quality model.

Definition 4.16 Possibilities for Impacts of a Quality-Property. *The relation* $\mathrm{IMP}_{\mathrm{all}} \subseteq \mathrm{P} \times \wp(\mathrm{P})$ *describes for each quality-property* p *the set of quality-properties* I *by which it can be impacted. A quality-property* p *is impactable by a set of quality-properties* I*, written as* $(p,\, I) \in \mathrm{IMP}_{\mathrm{all}}$*, if and only if:*

1. *The quality-property* p *is not impacted by the empty set:*

   $I \neq \varnothing$

2. *The domain of the quality-property* p *is a container of the domains of all quality-properties of* I*:*

   $\forall p_i \in I: \;\; (\mathrm{dom}(p),\, \mathrm{dom}(p_i)) \in \mathrm{COMP}^*$

3. *There is a function* $f : [0,1]^n \to [0,1]$ *which calculates an estimate of the impacted quality-property, using the values of the impacting quality-properties. For each impacting quality-property* $p_i \in I$*, this function describes either a* positive *or a* negative *dependency. A positive dependency means that if the impacting quality-property increases, then also the impacted quality-property increases, given that all other impacting quality-properties remain the same. A negative dependency is defined in analogy.*

$$
\begin{array}{l}
\exists f: \quad \forall p_i \in I: \\
\left(\begin{array}{l}
\forall (o,\, o_1,\, \ldots,\, o_n) \in \mathit{suitables}(p,\, I): \\
\forall (o',\, o_1,\, \ldots,\, o_{i-1},\, o'_i,\, o_{i+1},\, \ldots,\, o_n) \in \mathit{suitables}(p,\, I): \\
\qquad p(o) \; > \; p(o') \quad \Leftarrow \quad f(p_1(o_1), \ldots, p_i(o_i), \ldots, p_n(o_n)) \; > \\
\qquad\qquad\qquad\qquad\qquad\quad\; f(p_1(o_1), \ldots, p_{i-1}(o_{i-1}), p_i(o'_i), p_{i+1}(o_{i+1}), \ldots, p_n(o_n))
\end{array}\right) \begin{array}{c}\text{positive}\\ \text{dependency}\end{array} \\
\vee \\
\left(\begin{array}{l}
\forall (o,\, o_1,\, \ldots,\, o_n) \in \mathit{suitables}(p,\, I): \\
\forall (o',\, o_1,\, \ldots,\, o_{i-1},\, o'_i,\, o_{i+1},\, \ldots,\, o_n) \in \mathit{suitables}(p,\, I): \\
\qquad p(o) \; < \; p(o') \quad \Leftarrow \quad f(p_1(o_1), \ldots, p_i(o_i), \ldots, p_n(o_n)) \; > \\
\qquad\qquad\qquad\qquad\qquad\quad\; f(p_1(o_1), \ldots, p_{i-1}(o_{i-1}), p_i(o'_i), p_{i+1}(o_{i+1}), \ldots, p_n(o_n))
\end{array}\right) \begin{array}{c}\text{negative}\\ \text{dependency}\end{array}
\end{array}
$$

$$
\begin{array}{lll}
\textit{whereby} \quad \mathit{suitables}(p,\, I) & = & \{(o,\, o_1,\, \ldots,\, o_n): \; o \in \mathrm{dom}(p) \,\wedge\, o_i \in \mathrm{dom}(p_i) \,\wedge\, (o,\, o_i) \in \mathrm{COMP}^*_{\mathrm{U}}\} \\
\textit{and} \quad \{p_1,\, \ldots,\, p_n\} & = & I
\end{array}
$$

Definition 4.17 Impact Relation. *The impact relation* $\mathrm{IMP} \subseteq \mathrm{P} \times \mathrm{P}$ *syntactically defines the impacts that are considered in the quality model. It is consistent with the* $\mathrm{IMP}_{\mathrm{all}}$ *relation:*

$$
\forall p \in \mathrm{P}: \;\; \forall I \subseteq \mathrm{P}: \quad \left(\begin{array}{lll} & \forall p_i \in I: & (p, p_i) \in \mathrm{IMP} \\ \wedge & \forall p_i \in \mathrm{P} \backslash I: & (p, p_i) \notin \mathrm{IMP} \end{array}\right) \;\Rightarrow\; (p, I) \in \mathrm{IMP}_{\mathrm{all}}
$$

Because the impact relation is closely aligned with the composition relation on classes, the characteristic of acyclicity is transferred from the composition to the impact relation, as stated in the following theorem.

Theorem 4.5 Impact Relation is Acyclic.

Proof. *This follows from Definition 4.16, Definition 4.17 and from the acyclicity of the composition relation (Axiom 4.5).*

In the definition of $\mathrm{IMP}_{\mathrm{all}}$, we see that there is either a positive or negative dependency between each impacting and impacted quality-property. If there is a positive dependency, then we say the impact has a *positive effect*, if there is a negative dependency, we say the impact has a *negative effect*. A positive effect means that the degree of satisfaction of the target quality-property increases if the degree of satisfaction of the source quality-property increases. For a negative effect, the satisfaction of the target quality-property decreases if the satisfaction of the source quality-property increases.

Definition 4.18 Effect of an Impact. *The effect of an impact is a function* effect : $\mathrm{IMP} \rightarrow \{\oplus, \ominus\}$, *assigning a* $\oplus$ *if there is a* positive dependency *and assigning a* $\ominus$ *if there is a* negative dependency*, according to the definition of dependencies in Definition 4.16.*

For writing an impact $(p_s, p_t) \in \mathrm{IMP}$ we use the following notation: We write $p_s \xrightarrow{\oplus} p_t$ for an impact with a positive effect and $p_s \xrightarrow{\ominus} p_t$ for an impact with a negative effect.

In the quality model, we describe an impact and its effect usually by informal means; i.e., we give a description in prose to provide a rationale for an impact and its effect. This way of describing impacts is adequate for the information that is usually modeled by impacts. Ideally, empirical information will confirm that a quality-property has an impact on another quality-property. In this case, a statistical correlation has usually been discovered. A statistical correlation is a special case of the dependency relation defined in Definition 4.16, whereby a correlation describes a linear dependency. A correlation with a positive correlation coefficient maps to an impact with a positive effect, while a negative correlation coefficient maps to an impact with a negative effect. If no statistical evidence for an impact is available, then a rationale in prose is given, which explains why we assume a dependency to be present.

The impact relationship describes a dependency between quality-properties without quantifying it. For conducting actual quality assessments in Chapter 5, however, a quantitative specification of the relation between impacting quality-properties is needed. Hence, in that chapter we introduce a method for estimating the satisfaction of a target quality-property based on the impacting quality-properties. In other words, we define a method for specifying a concrete implementation of the function f of Definition 4.16.

4.4.5 Types of Quality-Properties

For achieving a clearer structure of the quality-property hierarchy, we split the quality-properties into two types. First, all quality-properties describing characteristics of the entire software product are called *quality attributes*. All other quality-properties, which refer to parts of the software product are called *component properties*.

Definition 4.19 Types of Properties. *The set of all quality-properties* P *is split into two sets: (1) quality attributes* P_{qa} *and (2) component properties* $\mathrm{P}_{\mathrm{component}}$. *The quality attributes have* $\mathrm{product}$ *as defined in Definition 4.7 as domain, while component properties do not have* $\mathrm{product}$ *as domain:*

$$\begin{aligned}\mathrm{P} &= \mathrm{P}_{\mathrm{qa}} \uplus \mathrm{P}_{\mathrm{component}}\\ \mathrm{P}_{\mathrm{qa}} &= \{p \in \mathrm{P} : \ \mathrm{dom}(p) = \mathrm{product}\}\\ \mathrm{P}_{\mathrm{component}} &= \{p \in \mathrm{P} : \ \mathrm{dom}(p) \neq \mathrm{product}\}\end{aligned}$$

Quality attributes always have $\mathrm{product}$ as domain, because they describe quality-properties of the entire product. Because they all have the same domain, it is not possible that they are restricted by other quality attributes. The only possible relation between two quality attributes is the division relation, as stated in the following theorem.

Theorem 4.6 Quality Attributes are only in a Division Relation. *Between two distinct quality-properties* $p_1, p_2 \in \mathrm{P}_{\mathrm{qa}}$ *no restriction and no impact relations exist; the only possible relation between them is the division relation.*

Proof. *Two quality-properties in* P_{qa} *always have* $\mathrm{product}$ *as domain. Thus, their domains cannot be in a generalization or composition relation. Since the restriction relation (Definition 4.11) requires the domains to be in a generalization relation, the two quality-properties cannot be in a restriction relation. Since the impact relation (Definition 4.17) requires the domains to be in a composition relation, the two quality-properties cannot be in an impact relation.*

Figure 4.6 shows the structure of the quality-property hierarchy. The hierarchy is split into two parts, the quality attribute hierarchy and the component property hierarchy. Within the quality attribute hierarchy only division relations are present. Within the component property hierarchy restriction and division relations are present. Formally, impact relations within the component property hierarchy are not excluded, but usually the component properties are chosen in a way that no such relations are there. Hence, the impact relations usually originate from the component property hierarchy and target the quality attribute hierarchy.

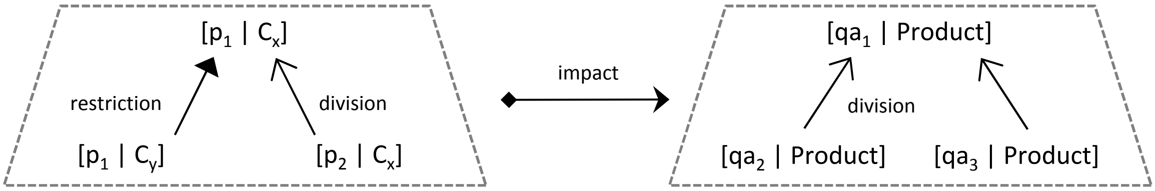


Figure 4.6: Structure of the Quality-Property Hierarchy

4.5 Summary

In this chapter, we introduced a quality meta-model for quality models of software. A quality model has the purpose of describing information on software quality in a way that it is usable for the different tasks of quality assurance. Thus, it must be able to integrate a large variety of information coming from different sources. This information must be organized in a way that it is understandable and manageable by people who use the quality model. Moreover, it should allow the integration of measurements in order to conduct quality assessments.

Hence, the main challenge in constructing a quality model is to find an adequate way of structuring all quality-related information. Our approach integrates a product model of software, which serves as a basis for defining quality-properties. It is easier for people to develop and understand a product model than to directly work on quality-properties. Furthermore, the product model defines the hierarchical structure of software, which is needed for quality assessments.

Based on the product model, our quality meta-model is built around one main concept for describing quality-related information, called *quality-property*. While in general a *property* is an arbitrary attribute of an object, a quality-property in our quality model is of quantitative nature, i.e., a quality-property quantifies the degree to which an object exhibits the quality-property on a continuous scale between 0 and 1. We use quality-properties to model quality characteristics on all levels of detail, starting from general quality attributes like *maintainability*, ranging to technical characteristics like *nesting depth of methods*.

Summing up, a quality model, according to our meta-model is essentially a collection of a large number of quality-properties. In order to structure these quality-properties and thereby make them manageable, we introduce several relations between quality-properties. The *restriction* relation describes a taxonomical arrangement of the quality-properties that is strongly aligned with the product model of software. The *division* relation is a taxonomical relation describing how quality-properties on one level of detail can be broken down into several sub-properties. The *impact* relation describes a causal influence of the quality-properties of components of a software system on general quality attributes.

Our quality meta-model defines all its concepts and relations in a formal way. Thereby, the semantics are well-defined and the problem of having unclear, ambiguous, and overlapping definitions in quality models (cf. Chapter 3) is avoided. The taxonomical structures introduced by the restriction and division relation are mainly used for presenting the quality model to people, by offering different views in tools (cf. Chapter 7). By providing structured views, the information captured in a quality model is made accessible and understandable to users. The impact relation captures information of prime importance: It describes how detailed technical quality-attributes relate to high-level quality attributes. This information is important for the informal usage by explaining why low level quality-properties are important for quality. Moreover, the aggregation of low-level measurement results is also done alongside the impact relations, leading to understandable and traceable quality assessment results (c.f. Chapter 5).

5 Quality Assessments based on a Quality Model

In Section 2.2 we discussed that the goal of analytic quality assurance is analyzing a product or process to test its conformance to requirements. In this chapter, we take an even narrower view and focus on a *product quality assessment*, which has the task of testing the software product for conformance to specified requirements.

In the literature, a wide range of quality assessment approaches is proposed, ranging from dynamic tests to manual inspections and automatic analysis of artifacts. The quality assessment approach presented in this chapter was developed with the usage scenario of static code analysis in mind. Thus, it is fitted for statically analyzing artifacts, with a strong focus on automated analysis and assessment. In the case study of Chapter 8 it is applied to static code analysis of Java source code. However, its principles may also be applied to other artifacts that can be statically analyzed.

Motivation

As outlined in Section 2.2.5, a quality model plays a substantial role for capturing requirements to a software system. It is used as a repository of requirements which are communicated to the development. According to the quality control cycle, the developed product must be tested for the satisfaction of requirements. The identified deviations from the specification are then communicated to the developers for correction.

For effectively establishing such a quality control, the frequency of the quality assessments must be high enough to enable a continuous feedback to the developers. A prime way of achieving a high frequency of quality assessments is to minimize the effort they require, which can best be achieved by automating them. The artifacts best suited for automated analysis are formalized, machine-readable ones, such as the source code of software.

Another motivation for using quality models as the basis for quality assessments is to achieve comparability between different assessments. A quality model precisely defines the properties relevant for high quality. By augmenting such a quality model with a precise specification of how the satisfaction of the properties can be tested, repeatability and comparability between different quality assessments can be achieved.

Problem

The quality meta-model, as described in Chapter 4, defines quality-properties of software artifacts and their relations. The quality-properties are formalized as functions for precisely specifying what the restriction and division of them means. The impact relation is specified by defining that two quality-properties are positively or negatively correlated, which does not give quantitative information regarding the influence of one quality-property on another. Furthermore, in concrete instances of the quality model, the quality-properties are only specified informally. Such an informal specification is sufficient for a precise definition of requirements, which are informally communicated to developers. However, it is not precise enough for testing whether a concrete software product satisfies these quality-properties.

For conducting a quality assessment, more precise information is needed. For instance, if the satisfaction of the quality-properties of the quality model should be checked by a manual review of the software, a detailed instruction is needed explaining how the satisfaction of quality-properties is being tested and on which scale the assessment has to be done. Furthermore, a detailed method for accumulating the single values to an overall quality statement is needed.

A fully automated quality assessment needs an even more formalized approach. It needs a machine-interpretable specification for evaluating the quality-properties and for aggregating the values to an overall quality statement.

Our Solution

Conducting a quality assessment with the quality model means determining whether a software product exhibits a quality-property defined by the quality model. Ultimately, the quality attributes like $[\mathit{Maintainability}|\mathit{Product}]$ are of interest for a quality assessment. However, these high-level quality-properties are not directly determinable. Thus, our approach is to estimate their satisfaction by other quality-properties. As discussed in Section 2.1, there are principally different ways of estimating these quality-properties. One way, although one not pursued in this work, would be to estimate them by measuring related processes. For instance, the maintainability could be assessed by measuring maintenance efforts during the maintenance process or the reliability could be measured by the frequency of failures in the field.

We pursue the approach of estimating high-level quality attributes exclusively by component properties. Using only the components of the product has several advantages: First, product artifacts can also be assessed before the product is finished and delivered to the customer. This way, an assessment based only on the product can be applied in earlier development stages. Second, measuring the product itself reduces the influence factors that could distort the quality assessment. For instance, maintenance efforts do not depend only on the product but also on organizational factors and on the people conducting the maintenance.

A quality model defines which component properties can be used to estimate a quality attribute: The impacts informally define the influence of detailed component properties on the high-level quality attributes. These impacts are usually informally justified or at best empirically confirmed. For an automated assessment of the quality attributes by its impacting quality-properties additional information has to be augmented. For instance, while the quality model defines that cloning of source code has a negative impact on maintainability, the aggregation specification has to define exact thresholds for acceptable cloning. Such definitions are generally less objective and less justifiable than the informal definitions of the impact. The definitions of thresholds often depend on technical issues, such as the technology or programming language under consideration.

Since we focus on using static analysis as the basis for quality assessments, we have to rely on existing methods and tools for statically analyzing software products. Thus, we have to define how static code analysis rules can be used for evaluating the satisfaction of quality-properties and how the data gathered thereby is used for evaluating an impacted quality attribute. Regarding this challenge, we developed a new method that specifically focuses on source code quality-properties.

Additionally, for actually applying an automated quality assessment relying on source code, some practical and partially technically-motivated challenges have to be overcome. We explain these challenges in detail and provide a solution for them.

Outline

The first section clarifies the basic concepts regarding the measurement of quality-properties. It also introduces a basic terminology on measurement and connects it to the terminology of the quality model. Then, the second section shows how measures and quality analysis tools are integrated into the quality model to operationalize quality-properties. Furthermore, it shows how the aggregation of measurement values, based on the structure of the quality model, takes place. The third section explains the specifics and limitations of real-world quality analysis tools. Moreover, it introduces a way for using these tools nonetheless.

The fourth section addresses three problems encountered when applying a quality assessment approach in practice. First, a common problem when using measures for quality assessments is to define utility functions, i.e., threshold values for measurement values. This problem is addressed by introducing a benchmarking approach for determining threshold values. Second, we introduce a method for supporting incomplete measurement data, i.e., conducting the quality assessment even though not all measurements are conducted. Usually, measurements are not conducted because the tools realizing them are not available in a specific setting. This is especially important to apply the approach in real-world industrial settings. Third, we take a closer look at rule-based static code analysis tools and how they can be integrated into the quality assessment approach.

5.1 Basic Concept

According to the quality model, the actually interesting concepts are quality-properties. In general, quality attributes are expressed as quality-properties of the class $\mathrm{product}$. So, conducting a quality assessment means evaluating the satisfaction of a quality-property.

A quality-property p with domain $\mathrm{dom}(p)$ can be evaluated for each object $o \in \mathrm{dom}(p)$. However, the quality model, does not give a specification as to how the function p is actually calculated. It only uses the notion of functions to specify how different quality-properties relate to each other.

The actual calculation of a quality-property is carried out by measures which are implemented by measurement tools. A measure is a more general function than a quality-property, but defined on the same domain.

Definition 5.1 Measure. *A measure for a quality-property $p \in \mathrm{P}$ is a function assigning values to the objects of the domain of the quality-property.*

$$\mathrm{measure}: \; \mathrm{dom}(p) \rightarrow \mathrm{M}$$

whereby M is a set of values, typically having some specific characteristics (see the definition of *scales* and different types of scales in the following paragraphs).

In software engineering, *metric* is often used as a synonym for both *measure* and *measurement value*. We do not use this term in this thesis.

The set M can have different characteristics, which determine the interpretation and the operations that can be performed on it. These characteristics are called scale.

Definition 5.2 Scale. *A scale describes the characteristics of the set* M *used in measurement.*

Usually, there are four types of scales used [143]:

- Nominal Scale (operations: equality)
  A scale which only defines equality of values of M. The elements of M are arbitrary values, like numbers, words or letters [143]. An example for this scale is “the ‘numbering’ of football players for the identification of the individuals” [143].

- Ordinal Scale (operations: rank-order)
  A nominal scale which defines a rank-ordering of the values of M. It describes a linear order on the values, but not the relative size of degree of difference between the values. An example is the scale of mineral hardness, which characterizes the hardness of minerals by the ability of a harder material to scratch a softer one, without defining the actual hardness in absolute terms.

- Interval Scale (operations: equality of intervals)
  An ordinal scale which defines the size of the intervals between the symbols. Typical examples are the temperature scales Fahrenheit and Celsius. Equal intervals of temperature denote equal volumes of expansion. However, the zero in an interval scale is arbitrary. Hence, ratios between numerals are meaningless on this scale. For instance, it is meaningless to say that 40°C is twice as hot as 20°C.

- Ratio Scale (operations: equality of ratios)
  An interval scale which defines a true zero point. Thus, ratios on this scale are equal. Examples of this scale are scales for measuring length: inches or meters. On this scale ratios are meaningful; for instance, 6 m is twice as much as 3 m. Another example is the temperature scale of Kelvin, which defines an absolute zero.

In order to use a measure in the quality model for calculating a quality-property a mapping function from M to $[0,1]$ has to be specified for each quality-property and measure. This mapping is called *fuzzification* [147], because it maps a value of an arbitrary scale to a truth value of fuzzy logic. A common example for fuzzification is to convert the measure “age in years” into a logical value of the property “young”.

Definition 5.3 Fuzzification. *Fuzzification means transforming a measurement on an arbitrary scale to a truth value of fuzzy logic. A fuzzification function is used for this task:*

$$\mathrm{map}: \ \mathrm{M} \to [0,1]$$

For examples of measures and a suitable fuzzification function for them, we refer to Section 5.2.2, where a comprehensive description of both these topics is given.

Measures in the quality model are concrete functions for which, unlike for quality-properties, a concrete implementation exists. Thus, by assigning measures to quality-properties and by providing a fuzzification function, the quality-properties can be calculated for actual objects. Providing executable specifications for quality-properties in form of measures is called operationalizing a quality model.

Definition 5.4 Operationalization. *Operationalizing a quality model means providing executable measures and fuzzification functions for quality-properties.*

The process of actually applying a measure to an object is called measurement and defined as follows.

Definition 5.5 Measurement. *Measurement is the process of assigning elements of a scale to an object according to the specification of the measure.*

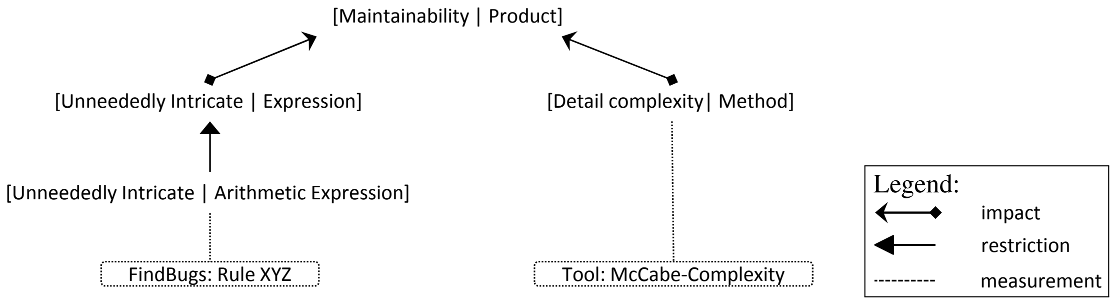
Definition 5.6 Measurement Data/Values/Results. *Measurement Data/Values/Results are the elements obtained by conducting a measurement.*

Definition 5.7 Measurement Tool. *A measurement tool is a technical implementation of a measure in the form of a software program.*

Thus, *executing a measurement tool* means *conducting a measurement*.

5.2 Top-Down Specification

In order to conduct quality assessments based on the quality model, the model must provide aggregation specifications, describing how measurement values are normalized and aggregated to an overall quality statement about a software product. The aggregation specifications are defined in a top-down manner: For a quality-property they specify how its value can be calculated using the values of its child quality-properties. These top-down specifications do not consider the restrictions of real-world measurement tools; the handling of these restrictions will be the subject of the subsequent sections.


[Maintainability | Product]
[Unneededly Intricate | Expression]
[Detail complexity| Method]
[Unneededly Intricate | Arithmetic Expression]
FindBugs: Rule XYZ
Tool: McCabe-Complexity
Legend:
impact
restriction
measurement

(a) Example Quality Model

$$
\begin{array}{l}
[Maintainability|Product]\,(MyProduct) = \\
\mathrm{aggr}_1(\\
\quad \mathrm{aggr}_{11}(\\
\quad\quad [\textit{Unneededly Intricate|Expression}](MyArithExpr1), \\
\quad\quad [\textit{Unneededly Intricate|Expression}](MyArithExpr2) \\
\quad), \\
\quad \mathrm{aggr}_{12}(\\
\quad\quad [\textit{Detail Complexity|Method}](MyMethod) \\
\quad) \\
)
\end{array}
$$

(b) Example Aggregation

Figure 5.1: Example of an Aggregation of Measurement Values in a Quality Model

5.2.1 Example of a Top-Down Quality Assessment

In this section, by means of an example, we describe how a quality assessment is conducted based on our quality model. As an example, the quality model illustrated in Figure 5.1a is used. The quality-property hierarchy describes that [*Maintainability*|*Product*] is negatively influenced by [*Unneededly Intricate*|*Arithmetical Expression*], which is measured by a *FindBugs*-rule. The maintainability is also negatively influenced by [*Detail complexity*|*Method*], measured by the *McCabe*-complexity.

An assessment in the ideal world takes a top-down approach. This means, the "root" quality-property [*Maintainability*|*Product*] is evaluated for a concrete object, in our example *MyProduct*. As a quality-property is a function mapping objects to the degree of satisfaction, the quality-property is applied to the object:

$$[Maintainability|Product]\,(MyProduct) \tag{5.1}$$

This quality-property is calculated using the impacting quality-properties. According to Section 4.4.4 all *parts* of *MyProduct* need to be determined; in the example all expressions and all methods of *MyProduct* need to be determined. In this example, we assume *MyProduct* consists of two arithmetical expressions and one method. Then the calculation is composed as shown in Figure 5.1: The quality-property [*Unneededly Intricate*|*Expression*] is determined for both arithmetic expressions and the quality-property [*Detail Complexity*|*Method*] is determined for the method. For each of these quality-properties, this results in a value for each object. These values are aggregated by the aggregation functions aggr_{ij}, resulting in one value per quality-property. These values are then aggregated by aggr_1 to one value for the impacted quality-property.

According to the definition of the restriction relation in Section 4.4.1 the two evaluations of $[\textit{Unneededly Intricate}|\textit{Expression}]$ follow the restriction relation to the quality-property $[\textit{Unneededly Intricate}|\textit{Arithmetical Expression}]$ and result in the following evaluations:

$$\begin{array}{l}[\textit{Unneededly Intricate}|\textit{Arithmetical Expression}](MyArithExpr1)\\ [\textit{Unneededly Intricate}|\textit{Arithmetical Expression}](MyArithExpr2)\end{array} \tag{5.2}$$

Both evaluations are determined using a measure realized by an external tool. The tool is invoked on *MyArithExpr1* and *MyArithExpr2* and returns the measurement values for the respective objects. These measurement values are then mapped to logical values of fuzzy logic. Additionally, the quality-property $[\textit{Detail complexity}|\textit{Method}]$ is determined using a measure. The external tool calculates the McCabe complexity for the method, which is also mapped to a logical value.

5.2.2 Specifying Measures

A value of a quality-property can be determined by a measure. Usually the leaf-quality-properties of the quality-property hierarchy are determined by measures. Measures can be realized either by measurement tools or manual inspections. In our quality model, we distinguish explicitly between three different types of measures:

1. Classical Measures
   A classical measure determines a numerical value for each object of the product. For instance, a classical measure could be "Complexity of Methods"; it determines a value for the complexity of each method. The measurement scale of such measures is arbitrary. Classical measures are implemented by measurement tools.

2. Rule-based Measures
   Rule-based measures describe rules for static code analysis. Such a measure applies rules to objects of the product, with the result of the rule being either violated or not. Thus, the measurement scale of such measures is nominal. Rule-based measures are implemented by static code analysis tools, which are a special type of measurement tools.

3. Manual Measures
   Instead of using automated tools for measurement, manual inspections by experts are also a way of measurement. The scale of a manual measure can be arbitrary. An expert can, for instance, assess the percentage of faulty constructs in a program, or rate objects on a scale "high", "medium", and "low".

Fuzzification

If a measure is used to determine the value of a quality-property, the measurement values (on an arbitrary scale) must be translated to a logical value of fuzzy logic. In the following, we discuss how the fuzzification is done for the measurement values in the quality model. We principally distinguish between two ways of fuzzifying measurement values: (a) automatically using a mapping function and (b) manual.

Automated Fuzzification of Classical and Manual Measures

As explained in Definition 5.1, measures provide a value on an arbitrary measurement scale, represented by a set M. According to Definition 5.3, the fuzzification of such a value means that a mapping function from M to $[0,1]$ has to be defined. For nominal, ordinal, and interval scales a specialized mapping function for each measure is required.

For ratio scales, usually a combination of two linear functions, as illustrated in Figure 5.2, is sufficient to express such a mapping function. For this function the quality model must define the three threshold values t_1, t_2, and t_3.

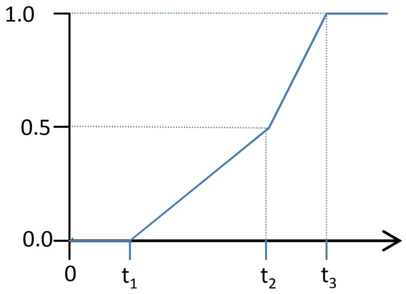


Figure 5.2: Mapping function with two linear segments

We illustrate the usage of this mapping function for the quality-property $[$*Many parameters*$|$*Concrete method*$]$. For measuring this quality-property a tool is used, calculating the number of parameters for each method of a software product, meaning $\mathrm{M} = \mathbb{N}_0$. For instance, the following thresholds are then used: $t_1 = 3$, $t_2 = 7$, and $t_3 = 9$. This means: Methods with less than or equal to three parameters evaluated to a degree of satisfaction of 0.0; methods from 3 to 7 parameters evaluated to values between 0.0 and 0.5; methods with 7 to 9 parameters evaluated to values between 0.5 and 1.0; methods with more than 9 parameters evaluated to the degree of satisfaction of 1.0 for the quality-property *Many parameters*.

Other mapping functions are also imaginable. In principle, for each measure an arbitrary mapping function may be defined. For instance, the complexity of concrete methods could be measured using the cyclomatic complexity according to McCabe [108]. It calculates a value for each method in the domain $\mathbb{N}_+$. A mapping function for mapping this scale to $[0,1]$ could be $\mathrm{map}(x) = 1 - \frac{1}{x}$. A method with the lowest cyclomatic complexity of 1 would be evaluated to the degree of satisfaction of 0.0; for increasing values of the cyclomatic complexity the degree of satisfaction converges to 1.0.

Automated Fuzzification of Rule-based Measures

As described earlier, rule-based static code analysis tools provide measurement data on a nominal scale: $\mathrm{M} = \{violated, not\text{-}violated\}$. For this scale, usually the following mapping function is used:

$$\mathrm{map}(x) \mapsto \begin{cases} 1 & \text{if } x = violated \\ 0 & \text{if } x = not\text{-}violated \end{cases} \tag{5.3}$$

According to this mapping, objects for which a rule was violated are evaluated to a value of 1, while objects without rule violations are evaluated to 0.

An example for a measure providing findings is the tool *JavaDocAnalysis* of *ConQAT*, which checks for methods without Java-Doc comments in Java source code. This tool reports all methods without a Java-Doc comment and is modeled as a quality-property [*Missing Java-Doc comment*|*Method*]. All methods without a Java-doc comment are evaluated to 1, meaning that the quality-property is fully satisfied; all methods with a comment are evaluated to 0.

Manual Fuzzification

When manually fuzzifying measurement data, the measurement data is presented to a human expert who assesses it and manually assigns the logical value. Manual fuzzifications are especially important to evaluate measurement data with a high fraction of false positives. In this case, the results provided by a measure are presented to an expert, who manually checks the results for validity. Depending on his judgment, the expert assigns a value to the quality-property.

An example for a measure where a semi-manual evaluation is advisable is the FindBugs-rule "DLS_DEAD_LOCAL_STORE", which is used to measure the quality-property [*Dead store to local variable*|*Assignment statement*]. According to the description of this rule, it very often produces false positives due to technical constraints of the analysis of the byte-code.

5.2.3 Specifying Restrictions

A value of a quality-property that is restricted by other quality-properties can be calculated using their values, given that the restriction is complete according to Definition 4.12. According to the definition of the restriction in Definition 4.11, no additional specification is needed in the quality model. Let p be a quality-property that is restricted by $p_1, p_2, \ldots, p_n$, then the quality-property p_i is chosen, whose domain contains the object o and whose domain is most specific:

$$\begin{array}{ll} (1) & p_{possible} = \{p_i \in \{p_1, \ldots, p_n\} : \; o \in \mathrm{dom}(p_i)\} \\ (2) & p_{specific} = \min_{\subseteq}(p_{possible}) \\ (3) & p(o) = p_{specific}(o) \end{array} \tag{5.4}$$

This calculation is always possible because the following holds:

1. $p_{possible}$ contains at least one element, if $\mathrm{complete}_{\mathrm{RES}}(p)$ is satisfied.
2. That $p_{specific}$ is unique, follows from Definition 4.11 and Axiom 4.3.

The value of the quality-property $p_{specific}$ must be determinable either by being divided into sub-properties or by being measured.

5.2.4 Specifying Divisions

A value of a quality-property that is divided into other quality-properties can be calculated using their values (see Definition 4.14). For each division relation, a function $f : [0,1]^n \to [0,1]$ exists that calculates the value of a quality-property based on the dividing quality-properties. The aggregation specification in the quality model must specify this function explicitly.

Principally, arbitrary functions can be used as aggregation functions. However, in practical quality models the following functions are typically used.

Logical Operators

Logical operators and especially all fuzzy logical operators (see Section A.2) can be used as aggregation functions. Mostly, four standard aggregation functions based on logical operators are used:

1. $f(v_1, \dots, v_n) = v_1 \wedge \dots \wedge v_n$
2. $f(v_1, \dots, v_n) = v_1 \vee \dots \vee v_n$
3. $f(v_1, \dots, v_n) = \neg v_1 \wedge \dots \wedge \neg v_n$
4. $f(v_1, \dots, v_n) = \neg v_1 \vee \dots \vee \neg v_n$

Statistical Operators

Other commonly used aggregation functions are statistical functions such as minimum, maximum, mean, median, and quantiles.

Weighted Sum

Beside the statistical functions, a weighted sum can also be used:

$$f(v_1, \dots, v_n) = w_1 \times v_1 + \dots + w_n \times v_n \qquad \text{with } \textstyle\sum_{i=1..n} w_i = 1 \tag{5.5}$$

The weights w_i for the sub-quality-properties have to be set by the quality modeler.

Ranking-based Weighted Sum

When using the weighted sum for aggregation, the quality modeler must set the weights of the sub-quality-properties. However, setting weights in an expert-based way is a challenging task. In decision theory much work on expert-based weighting of attributes has been done [7,135,165,170]. Multi-attribute decision models describe the influence of a set of attributes on a decision. The goal of such models is capturing the influence of the single attributes on the decision. A key component of such models is therefore the selection of weights for the single attributes. Eliciting the weights directly from experts leads to unsatisfactory results [135,165]. The weights elicited directly strongly depend on the elicitation method and particularly on the chosen range of weights [165].

Quality-Property	Rank	Weight
[*Too long*|*Concrete method*]	1	0.6111
[*Too deeply nested*|*Concrete method*]	2	0.2778
[*Too many parameters*|*Concrete method*]	3	0.1111

Table 5.1: Weights Determined Based on Ranks

Therefore, indirect methods of determining weights for attributes have been developed. A common method is that an expert ranks the attributes according to their importance. Then, the weights are calculated based on the given ranking. Again, for calculating the weights based on the ranks, different algorithms are available [7, 60, 144]. Different studies show that the rank order centroid (ROC) method of Barron [60] works best [7, 130]. They performed various case studies and concluded that it is the most reliable one and that results based on it reflect the intuitive estimation better than other approaches. Beside the fact that it is hardly possible to elicit direct weights, eliciting rankings has further advantages. Eckenrode [35] (cited in [7]) discovered that eliciting weights is easiest and produced repeatable outcomes. Furthermore, ranks can be elicited with less effort and in group discussions it is easier to agree on ranks than on exact weights [83].

Therefore, we use the ROC method for deriving weights from ranks. In the quality model ranks for all sub-quality-properties are defined. Then, a weighted sum based on the weights derived from the ranks is calculated.

Using the example of Section 4.2 (Figure 4.3), we show in Table 5.1 how ranks and weights for the sub-quality-properties of [*Complexity*|*Concrete method*] are defined.

5.2.5 Specifying Impacts

A value of a quality-property that is impacted by other quality-properties can be calculated using their values. Figure 5.3 shows the structure of such a calculation: In the quality model, a quality-property p may be impacted by several quality-properties $p_1, \ldots, p_n$. Unlike in the case of restrictions and divisions, each impacting quality-property p_i is evaluated regarding multiple objects $o_{i1}, \ldots, o_{im}$.

Hence, the evaluation function for impacts can be expressed as a calculation in two stages. First, for each impact all objects are evaluated regarding the impacting quality-property. The resulting values

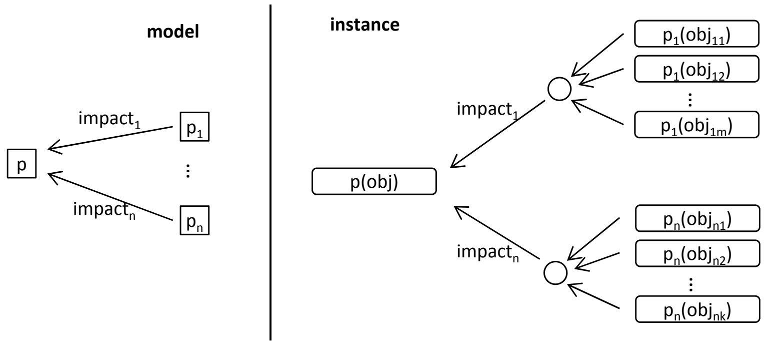


Figure 5.3: Evaluation of an Impact

of all these objects are then aggregated to one value. Second, all aggregated values (one value per impact) are aggregated to get the result value:

$$
\begin{aligned}
p(o) = \quad & \mathrm{aggr}_{impacts}(\\
& \quad \mathrm{aggr}_{impact_1}(\\
& \qquad p_1(o_{11}), \\
& \qquad p_1(o_{12}), \\
& \qquad \ldots \\
& \qquad p_1(o_{1m}) \\
& \quad), \\
& \quad \ldots \\
& \quad \mathrm{aggr}_{impact_n}(\\
& \qquad p_n(o_{n1}), \\
& \qquad p_n(o_{n2}), \\
& \qquad \ldots \\
& \qquad p_n(o_{nk}) \\
& \quad) \\
&)
\end{aligned}
\tag{5.6}
$$

An additional challenge in this aggregation is that all values of aggr_{impact_i} must be of the same kind in order to be meaningfully aggregated by $\mathrm{aggr}_{impacts}$. The following example illustrates the problem: The quality-property $[$*Analyzability*$|$*Product*$]$ is impacted by both $[$*Unnecessarily complicated*$|$*Comparison expression*$]$ and $[$*Detail complexity*$|$*Java Class*$]$. Despite both quality-properties being evaluated on fuzzy logical values, it does not make sense to just aggregate them to a value for the analyzability of the product, because they are each concerned with different classes. It is, for instance, not clear if a complex Java class is as bad for analyzability as a complicated comparison expression. Before aggregating these two values, they have to be converted to be of the same kind. Hence, the functions aggr_{impact_i} must all yield results of the same kind.

Specifying $\mathrm{aggr}_{impacts}$

Given that all functions aggr_{impact_i} provide values of the same kind, for realizing the function $\mathrm{aggr}_{impacts} : [0,1]^n \rightarrow [0,1]$ the same aggregations as for divided quality-properties can be used (see Section 5.2.4).

While, for divisions, weighted sums are often sufficient, for impacts they are not, because even a small proportion of unsatisfied impacting quality-properties may impair the target quality-property. For instance, the quality-property $[$*Maintainability*$|$*Product*$]$ may be influenced by 19 quality-properties which are satisfied and one quality-property which is not satisfied. This may lead to the dissatisfaction of $[$*Maintainability*$|$*Product*$]$. An example of quality-properties with such an influence are $[$*Cloning*$|$*Source code*$]$ and $[$*Conciseness*$|$*Identifiers*$]$. If a system contains a large number of clones, its maintainability is bad, independent of other characteristics. The same applies for the conciseness of identifiers: Inconcise identifiers render a program hard to understand, independent of other characteristics.

In fuzzy logic (see Section A.2), operators and modifiers suited for expressing this type of relation between variables are known. Usually one of the following operators is applied: (a) compensatory lambda operator, (b) compensatory gamma operator, or (c) weighted sum transformed by gamma modifier.

Specifying aggr_{impact_i}

The functions aggr_{impact_i} must provide values of the same kind. The simplest form to obtain values of the same kind is to express the differences by weights. Hence, to each object a weight is assigned and a weighted sum is calculated:

$$\begin{aligned}\mathrm{aggr}_{impact_i} &: (\mathrm{U} \times [0,1])^n \rightarrow [0,1] \\ \mathrm{aggr}_{impact_i}(o_1, v_1,\, o_2, v_2,\, \ldots,\, o_n, v_n) &\mapsto \frac{\sum_{i\,=\,1,\ldots,\,n} v_i * \mathrm{weight}(o_i)}{\sum_{i\,=\,1,\ldots,\,n} \mathrm{weight}(o_i)}\end{aligned} \tag{5.7}$$

The determination of a weight for each object is realized by a function $\mathrm{weight} : \mathrm{U} \rightarrow \mathfrak{R}$. For each class of objects a dedicated weight-function must be defined. However, for source code we can define a general weight function that can be applied to various source code constructs. This weighting is called *weighting by ranges* and is explained in the following.

Specifying aggr_{impact_i} – Weighting by Ranges

As explained in Section 5.2.2, quality-properties measured by rule-based static-code measures have a range of $\{0,\ 1\}$. If such quality-properties are restricted or divided and aggregated by logical functions their parent quality-properties also have a range of $\{0,\ 1\}$. For such quality-properties a special type of weighting is often reasonable, which takes into consideration which proportion of the source code is concerned with the respective quality-property.

For such quality-properties, an object may get either a 1 (we call the object *not affected* by the quality-property), or the object may get a 0 for a quality-property (we call the object *affected* by the quality-property). The idea of determining a value for the impact is then to calculate the proportion of the size of affected parts of the source code in comparison to unaffected parts.

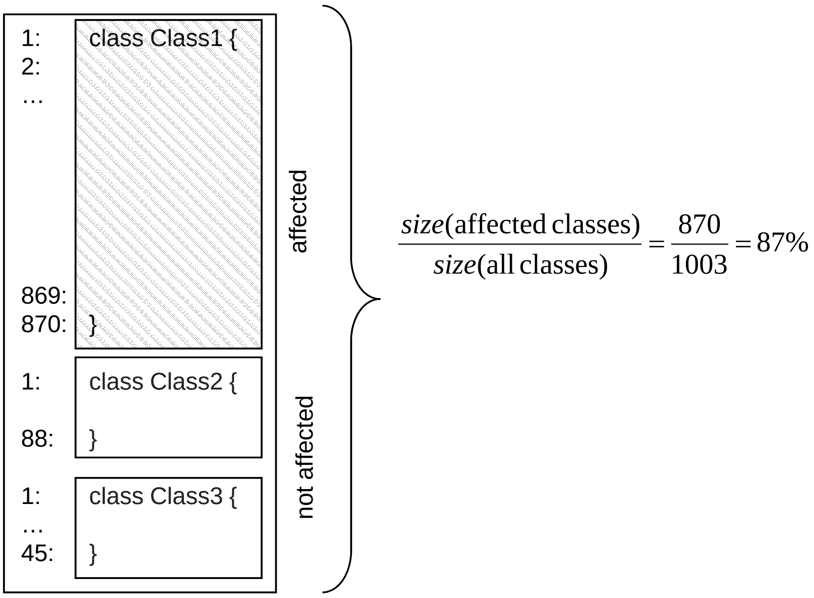


Figure 5.4: Example: Weighting ASccording to the Size of Java Classes

Example 1

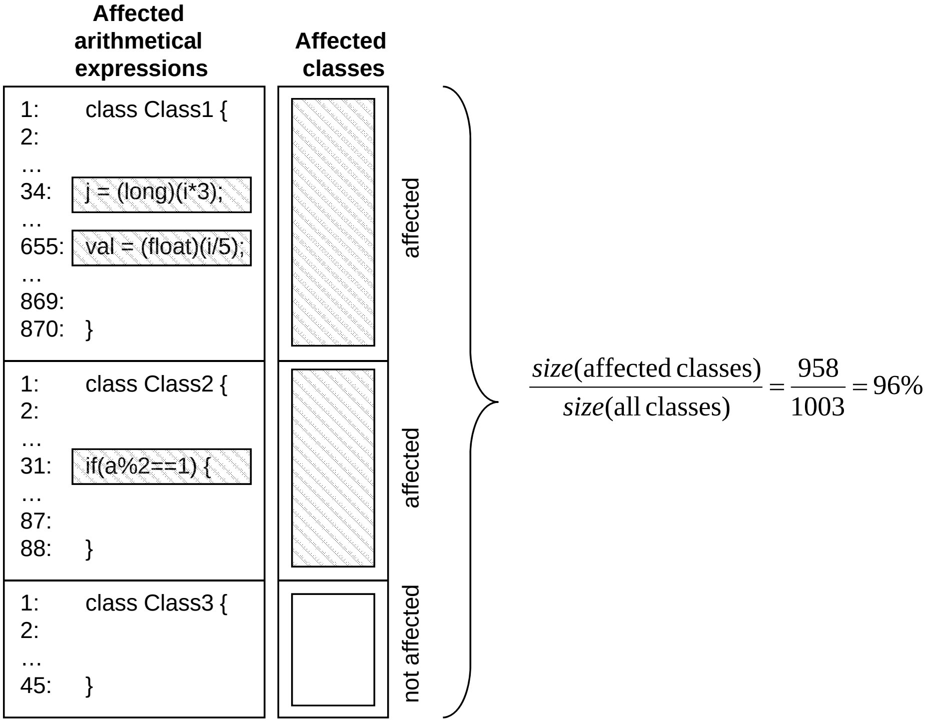
Figure 5.4 illustrates an example, where the impact $[\mathit{Appropriateness}|\mathit{Java\ class}] \xrightarrow{\oplus} [\mathit{Analyzability}|\mathit{Product}]$ is evaluated. First, for all objects of type “Java class”, the quality-property is calculated. This way, each Java class is either “affected” or “not affected”. Next, the size of each Java class is calculated and the evaluation of the impact is the proportion of the sizes of affected Java classes versus unaffected Java classes: $\mathrm{Size}\,(\text{affected java classes})\,/\,\mathrm{Size}\,(\text{all java classes})$.

This principle of calculating the size of affected vs. unaffected parts of the system works, if meaningful measures for the size of the objects under consideration can be found. For Java classes and methods, we use the lines of code as a measure of their size. We cannot apply this principle to other classes directly. For instance, it does not make sense to calculate the size in lines of code of the class *Arithmetical expression*.

Therefore, we extend our principle to so called *ranges*. If we want to calculate the impact of a quality-property, whose class has no meaningful size, we define one of its parent classes (regarding the composition relation) as its *range*. The class we choose as range must have a meaningful size. Therefore, we define that an object of the range-class is affected if it contains at least one object that is affected. This way, for each range-object it is defined whether it is affected or not and the principle of the weighting can be applied to the range object.

Example 2

The impact $[\mathit{Correctness}|\mathit{Arithmetical\ expression}] \xrightarrow{\oplus} [\mathit{Functional\ correctness}|\mathit{Product}]$ is calculated. *Arithmetical expression* is part of *Method*, which is part of *Java class*. We choose *Java class* as range of this impact. This means a Java class is *affected* if it contains at least one arithmetical expression that is not correct; and it is not affected if it contains no such arithmetical expression. Then,



```
1:    class Class1 {
2:
...
34:   j = (long)(i*3);
...
655:  val = (float)(i/5);
...
869:
870: }
```

```
1:    class Class2 {
2:
...
31:   if(a%2==1) {
...
87:
88:  }
```

```
1:    class Class3 {
2:
...
45:  }
```

$$\frac{size(\text{affected classes})}{size(\text{all classes})} = \frac{958}{1003} = 96\%$$

Figure 5.5: Example: Ranges

the principle of normalization is applied and the size of affected Java classes is set in proportion to the size of all Java classes. Figure 5.5 illustrates this.

Formal Definition of the Weighting by Ranges

In the following, we more formally define how an impact of a quality-property $[p_i|c_i]$ on $[p|c]$ is calculated by weighting with the range $c_r \in \mathrm{C}$.

Weighting by a range $c_r \in \mathrm{C}$ is possible for the given impact, if c_r is a part of c and a container of c_i:

$$(c, c_r) \in \mathrm{COMP}^* \quad \wedge \quad (c_r, c_i) \in \mathrm{COMP}^* \tag{5.8}$$

Furthermore, there must be a meaningful way of determining the size of objects of class c_r. This means there must be a function size for these objects:

$$\mathrm{size}: \; c_r \rightarrow \mathfrak{R} \tag{5.9}$$

Accordingly, the size of a set of objects is defined as the sum of the sizes of the sets' elements:

$$\begin{aligned} &\mathrm{Size}: \; \wp(\mathrm{U}) \rightarrow \mathfrak{R} \\ &\mathrm{Size}(objs) \; \mapsto \; \textstyle\sum_{obj \in objs}(\mathrm{size}(obj)) \end{aligned} \tag{5.10}$$

Now, we can formally define how the ratio of affected vs. unaffected parts of the product is calculated:

$$ratio = \frac{\text{Size}\left(\{o_i \in c_r : \ (o, o_i) \in \text{COMP}_\text{U} \ \land \ \exists o_j \in c_i : \ (o_i, o_j) \in \text{COMP}_\text{U} \ \land \ p_i(o_j) = 1\}\right)}{\text{Size}\left(\{o_i \in c_r : \ (o, o_i) \in \text{COMP}_\text{U}\}\right)} \tag{5.11}$$

Although the ratio is in the interval $[0; 1]$, it does not adequately represent a degree of satisfaction of the quality-property. Imagine a quality-property describing some kind of security vulnerability, having an impact on $[$*Security*$|$*Product*$]$. If the impact of such a quality-property gets a ratio of 0.05 this means, that 5% of the system is affected by a security problem. In this case, it is unreasonable to just take the inverse as degree of satisfaction for the impacted quality-property, because this would mean that $[$*Security*$|$*Product*$]$ is satisfied with a degree of 0.95. Obviously, a system where 5% of the source code has security vulnerabilities should not get a rating that security is satisfied by 95%.

In order to map the ratios to degrees, we use a three-point linear distribution as depicted in Figure 5.6. The function maps a normalized measurement value to a degree of satisfaction of a quality-property. Such functions are usually called *utility*-functions.

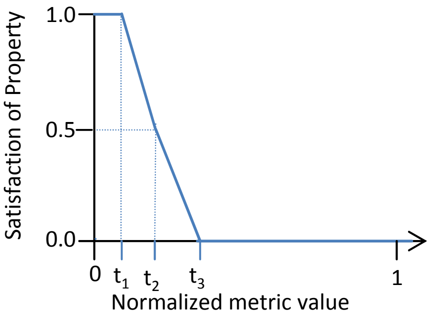


Figure 5.6: Three-point Linear Distribution Function

In order to obtain an identity function, the values t_1, t_2, t_3 are chosen as $t_1 = 0$, $t_2 = 0.5$, $t_3 = 1$. In realistic quality models these values will be chosen by either expert appraisal, or by a benchmarking approach (see Section 5.4.1 for details).

For the security-related example above, the values t_i could be chosen as follows:

$$t_1 = 0.05 \quad t_2 = 0.1 \quad t_3 = 0.2 \tag{5.12}$$

The resulting distribution expresses the fact that all systems that are affected more than 20% get a rating of 0.0. To achieve a rating better then 0.5, less than 10% must be affected, while to get a rating of 1.0, less than 5% must be affected.

5.3 Bottom-Up Assessment

In Section 5.2 we generally describe which specifications are necessary for arriving at an operationalized quality model. The weighting by ranges is inspired by the assessment of source code, but not explicitly restricted to it. In this section, however, we explicitly describe how an automatic

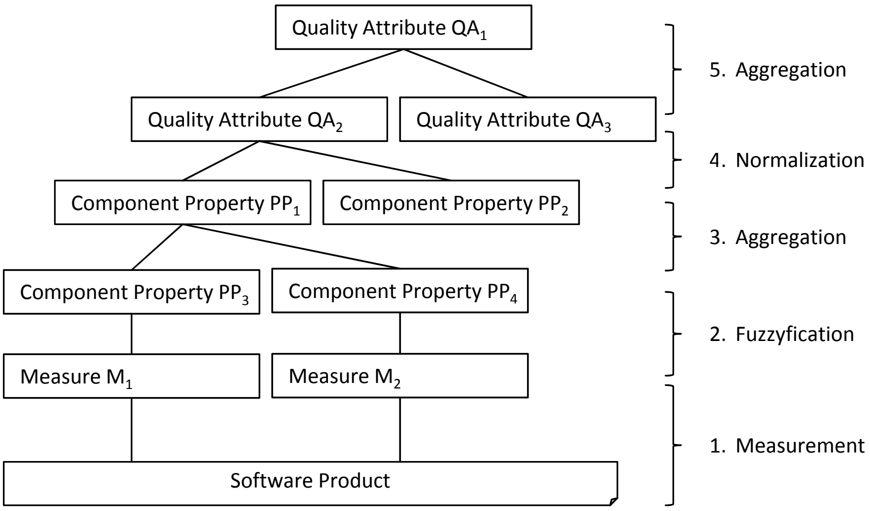


Figure 5.7: Steps of a Bottom-Up Assessment

tool-based assessment of source code can be done. Thus, we explain how typical tools for source code analysis work and how they can be integrated with the top-down specification approach for quality assessments. This approach takes a bottom-up form, for two reasons:

1. One reason is that existing tools are applied to whole products, not parts of products. They determine the relevant parts of the product and report the measurement results for them. Thus, the bottom-up approach runs the measurement tools first and then aggregates the resulting measurement data.

2. Another reason is that, for a top-down approach, the parts of the product must be determined, which is a challenging task. It strongly depends on the technology used, e.g., the programming language. Since the existing tools already implement this step, it is not necessary to re-implement it by the assessment approach. For real-world systems and quality models, it is unfeasible to implement a function for determining all objects of a certain type for all classes modeled in the quality model. For some general and commonly used classes, these functions are necessary for weighting by ranges; e.g., when analyzing source code it is necessary to determine the files, classes, and methods of which the source code consists. However, determining more special classes such as expressions, fields, constructors, or even Java-classes of certain super-types (e.g., all Java-classes implementing java.util.Iterator) is expensive. Furthermore, this would mean that each time a class is added to the quality model it would be necessary to adapt the assessment toolkit. Hence, the bottom-up approach uses the data on the objects of the product provided by the existing analysis tools.

In the following, we describe the bottom-up assessment approach. Figure 5.7 shows the steps of the bottom-up quality assessment:

1. *Measurement:* The first step in a quality assessment is to conduct the measurement. For that purpose the measurement tools are executed and manual inspections are conducted. There are two types of measurement tools for source code:
   Classical Measures are defined for single objects of a product. The measurement is executed

on the whole product and for each part of the product that is of a certain type a measurement value is returned. For instance, such a measure could be "Complexity of Methods"; when applied on the source code of a product, it determines a value for the complexity of each method.

Rule-based static code analysis tools have the objective of finding defects in source code. They analyze the entire source code of a product and report objects of the product that are faulty. This means they calculate a quality-property of all parts of the product and report those objects where the quality-property does not hold. The initial intention of these tools was to report these faulty objects, called *findings*, to the developer so that he can correct them.

2. *Fuzzification:* The measurement result of step 1 is fuzzified, i.e., converted to logical values of fuzzy logic.

3. *Aggregation:* In this step the values of component properties are aggregated along the restriction and division relations.

4. *Normalization:* At the transition from component properties to quality attributes via the impacts in the quality model it is specified how the weighting takes place.

5. *Aggregation:* In this step the values of the quality attributes are aggregated along the division relations.

The main difference from the procedure of the top-down approach is that the measurement tools are run on the entire product. Hence, they generate as a result a *list of all objects with their corresponding values*. The fact that a quality-property is evaluated on the entire product is formalized by a function eval.

Definition 5.8 Bottom-up Evaluation. *The bottom-up evaluation of an entire product regarding a quality-property $p \in \mathrm{P}$ is defined as a function* eval *resulting in a list of objects with their respective values:*

$$\mathrm{eval}: \ \mathrm{product} \times \mathrm{P} \ \rightarrow \ \wp(\mathrm{U} \times [0,1])$$

whereby $\mathrm{product}$ denotes the set of all products according to Definition 4.7 and P denotes the set of all quality-properties according to Definition 4.9.

The bottom-up quality assessment approach must guarantee that the results generated by it are the same as if the top-down approach had been applied.

Definition 5.9 Correctness of Bottom-up Evaluation. *A bottom-up evaluation of a quality-property $p \in \mathrm{P}$ conducted by the function* eval *is correct if its result contains all objects $o_i \in \mathrm{dom}(p)$ which are part-of the product under consideration:*

$$\begin{aligned}&\forall p \in \mathrm{P}: \ \forall product_i \in \mathrm{product}: \\ &\quad \mathrm{eval}(product_i, p) \ = \ \{(o, p(o)): \ o \in \mathrm{dom}(p) \wedge (product_i, o) \in \mathrm{COMP}_{\mathrm{U}}^{*}\}\end{aligned}$$

In the following, we will show that the aggregations specified in a top-down manner can be calculated on the *lists of objects and their values*.

Measures

As explained above, common measurement tools already implement the function eval for certain quality-properties. Thus, these tools can be used directly to calculate this function.

Restriction

Let p be a quality-property that is restricted by $p_1, \ldots, p_n$ and $\mathrm{eval}(product_i, p_i)$ the results of the evaluation of p_i regarding $product_i \in \mathrm{product}$ satisfying Definition 5.9. Then, $\mathrm{eval}(product_i, p)$ can be calculated as follows:

$$\mathrm{eval}(product_i, p) = \bigcup_{j=1..n} \mathrm{eval}(product_i, p_j) \tag{5.13}$$

Obviously, $\mathrm{eval}(product_i, p)$ satisfies Definition 5.9.

Division

Let p be a quality-property that is divided into $p_1, \ldots, p_n$. The definition of the division relation (Definition 4.14) states the existence of an aggregation function f. Further, let $product_i \in \mathrm{product}$ be a product and $res_j = \mathrm{eval}(product_i, p_j)$ be the results of the evaluation of p_j satisfying Definition 5.9. Then, we define

$$\begin{aligned} objects(res_1, \ldots, res_n) &= \left\{ o : \ (o, v) \in \bigcup_{j=1..n} res_j \right\} \\ value(res_j, o) &= v : \ (o, v) \in res_j \end{aligned} \tag{5.14}$$

Then we calculate $\mathrm{eval}(product_i, p)$ as follows:

$$\mathrm{eval}(product_i, p) = \left\{ \begin{array}{ll} (o, v) : & o \in objects(res_1, \ldots, res_n) \quad \wedge \\ & v \ = \ f(value(res_1, o), \ldots, value(res_n, o)) \end{array} \right\} \tag{5.15}$$

Obviously, $\mathrm{eval}(product_i, p)$ calculated this way also satisfies Definition 5.9.

Impact

Let p be a quality-property impacted by the quality-property p_i and $\mathrm{eval}(product_j, p_i) = \{(o_1, v_1), \ldots, (o_n, v_n)\}$ the results of the evaluation of p_i regarding $product_j \in \mathrm{product}$. Then, all values required for the aggregation function aggr_{impact_i} as specified in Formula 5.6 are available and the function can be calculated.

5.4 Addressing Challenges in Practice

5.4.1 Determining Parameters for Utility Functions

At two points in the quality model, based on a measurement value, a degree of satisfaction of a quality-property must be calculated: (1) for fuzzifying a measurement value (Section 5.2.2) (2) when applying the range-based weighting (Section 5.2.5).

As explained in the respective sections, a 3-point linear function is used for the mappings. Whether the function is falling or increasing is determined by expert appraisal. For the determination of the three threshold values t_1, t_2, and t_3, a benchmarking approach is used [50, 51, 94, 100]. The basic principle of benchmarking is to collect a measure for a (large) number of systems (called benchmarking base) and compare the measurement value of the system under assessment to these values. This allows us to decide if the system is better, equally good, or worse regarding the benchmarking base. Our approach uses the benchmarking base to statistically determine threshold values for the utility functions. In essence, as a lower bound for the utility function, the minimum value of all benchmarking systems is used and as an upper bound, the maximum of all benchmarking systems is used.

For each system of the benchmarking base, the measurement value is calculated, resulting in values $x_1, \ldots, x_n$. The values of the thresholds are then calculated as follows:

$$\begin{aligned} t_1 &= \min(\{x : \ x \geq Q_{25\%}(x_1, \ldots, x_n) - 1.5 \cdot \mathrm{IQR}(x_1, \ldots, x_n)\}) \\ t_2 &= \mathrm{median}(x_1, \ldots, x_n) \\ t_3 &= \max(\{x : \ x \leq Q_{75\%}(x_1, \ldots, x_n) + 1.5 \cdot \mathrm{IQR}(x_1, \ldots, x_n)\}) \end{aligned} \tag{5.16}$$

Whereby $\mathrm{IQR}(x_1, \ldots, x_n)$ denotes the inter-quartile-range $Q_{75\%}(x_1, \ldots, x_n) - Q_{25\%}(x_1, \ldots, x_n)$. This function assures that outlier values are ignored; for t_1 the minimum non-outliner value is taken; for t_3 the maximum non-outlier value. For t_2 the median of all values is taken.

Alves et al. [2] discuss essential characteristics of a method for deriving threshold values for measures. Our method satisfies these requirements: (a) our method does not make an assumption about the distribution of the measure; (b) when combining the measurement values of different systems, the size of the system does not matter, i.e., large systems do not have a larger influence than smaller systems; and (c) our method is resilient against outliers by using the maximum and minimum non-outliner values.

In the publication [100], we thoroughly evaluated this approach. We used benchmarking bases with different characteristics for setting the parameters of utility functions of a quality model and compared the quality assessment results using the quality models. We found evidence that the single most important factor for a satisfactory benchmarking base is its size. For a randomly generated benchmarking base of sufficient size, neither the actually selected systems, nor the size of the systems contained in it, has a major influence on the quality assessment results generated by the quality model. Thus, we concluded that defining parameters for utility functions based on a benchmarking approach is feasible.

5.4.2 Handling Incomplete Measurement Data

When quality assessments are conducted in practice, incomplete measurement data are a common phenomenon. There are different reasons why measurement data can be incomplete:

1. A common reason for incomplete data is that an analysis tool incorporated into the quality assessment did not run successfully. If a large number of tools are used, this occurs frequently. Producing a valid assessment result is nonetheless essential. Especially in quality dashboards built continuously, the failures of one tool must not invalidate the overall result.

2. Besides the arbitrary failing of tools in continuous builds, another reason may be that a certain tool is unavailable. If the quality model and assessment toolkit is delivered to different customers, certain tools may not be available to a specific customer. For instance, license costs may prohibit the usage of a certain tool. In this case, the assessment approach must be able to deliver valid results without the information of the missing tool.

3. Another reason for incomplete data may be results of manual inspections that have not been carried out. Manual inspections are usually associated with significant efforts to conduct them. Therefore, it is often desirable to conduct a quality assessment relying solely on automatically determinable measures. Also in this case, the assessment approach must provide valid results.

In order to handle incomplete data, an approach similar to interval arithmetic is taken. In interval arithmetic, uncertainties are expressed by performing calculations on intervals of possible values. Each interval expresses that a calculation with complete data could result in one value of the respective interval. Instead of intervals, we use arbitrary sets of values.

All calculations will be performed on $\wp([0,1])$ instead of $[0,1]$. A function $f: [0,1]^n \rightarrow [0,1]$ is converted to $f^\circ: \wp([0,1])^n \rightarrow \wp([0,1])$, as follows:

$$f^\circ(x_1, \ldots, x_n) \mapsto \{f(z_1, \ldots, z_n): z_1 \in x_1, \ldots, z_n \in x_n\} \tag{5.17}$$

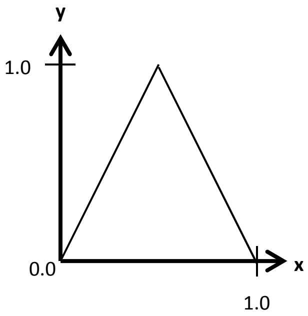
This way of handling incomplete data satisfies an essential requirement: If the approach capable of handling incomplete data is applied to complete input data, it must produce the same result. This can easily be proven: Let $f(x_1, \ldots, x_n) = y$ be an application of f to $x_1, \ldots, x_n$ with the result y. Then, f° applied to sets containing the values x_i each must result in a set containing exactly the value y:

$$\begin{aligned} & f^\circ(\{x_1\}, \ldots, \{x_n\}) \\ = \; & \{f(z_1, \ldots, z_n): z_1 \in \{x_1\}, \ldots, z_n \in \{x_n\}\} \\ = \; & \{f(x_1, \ldots, x_n)\} \\ = \; & \{y\} \end{aligned} \tag{5.18}$$

Example: If-Else-Expression

The approach of handling unknown values is applicable to sophisticated expressions, especially for non-continuous functions. For instance, a typical *if-then-else*-expression known from functional programming can be evaluated this way.

We want to realize the following function for transforming a value of $[0,1]$:



We implement it as a simple if-then-else-expression:

```
if(x > 0.5) then
    return (1 - x) * 2;
else
    return x * 2;
```

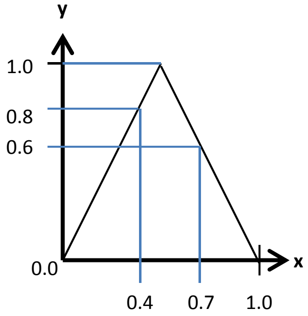
The condition $a > b$ is a function

$$\begin{aligned} \text{cond}: \quad & [0,1] \times [0,1] \to \mathbb{B} \\ \text{cond}(x,y) \quad & \mapsto \begin{cases} 1 & \text{if } x > y \\ 0 & \text{if } x \leq y \end{cases} \end{aligned} \tag{5.19}$$

The if-then-else construct is a function

$$\begin{aligned} & \text{ifthenelse}: \mathbb{B} \times [0,1] \times [0,1] \to [0,1] \\ & \text{ifthenelse}(\mathit{cond}, \mathit{thenres}, \mathit{elseres}) \mapsto \begin{cases} \mathit{thenres} & \text{if } \mathit{cond} = 1 \\ \mathit{elseres} & \text{if } \mathit{cond} = 0 \end{cases} \end{aligned} \tag{5.20}$$

We now evaluate this if-then-else construct with the value $a = \{0.4, \ldots, 0.7\}$, expressing that the value of a may be in the range between 0.4 and 0.7.



$$\begin{aligned} & \text{ifthenelse}^{\circ}(\mathit{cond}(a), (1-a)*2, a*2) \\ = \; & \{\text{ifthenelse}(\mathit{cond}(a_i), (1-a_i)*2, a_i*2) : a_i \in \{0.4, \ldots, 0.7\}\} \\ = \; & \{\text{ifthenelse}(1, (1-a_i)*2, a_i*2) : a_i \in \{0.5, \ldots, 0.7\}\} \cup \\ & \{\text{ifthenelse}(0, (1-a_i)*2, a_i*2) : a_i \in \{0.4, \ldots, 0.5\}\} \\ = \; & \{1-a_i)*2 : a_i \in \{0.5, \ldots, 0.7\}\} \cup \{a_i*2 : a_i \in \{0.4, \ldots, 0.5\}\} \\ = \; & \{0.6, \ldots, 1.0\} \cup \{0.8, \ldots, 1.0\} \\ = \; & \{0.6, \ldots, 1.0\} \end{aligned} \tag{5.21}$$

Here we can see that the result is as expected. For input values between 0.4 and 0.7, the function results in values between 0.6 and 1.0.

5.4.3 Handling Finding-Based Measurement Data

As indicated in the introduction of the bottom-up assessment approach in Section 5.3, rule-based static code analysis tools are a special challenge, for two reasons:

1. They are applied to the entire product and return a list of results.
2. They only report those objects violating a rule, not those satisfying a rule. Thus, the measurement data provided by them lack a list of all objects of the class under consideration.

In Section 5.3, we introduced a function eval evaluating all objects of a product regarding a certain quality-property. First, we apply the approach of handling incomplete measurement data on it, resulting in the following function:

$$\mathrm{eval}^{\circ}: \ \mathrm{product} \times \mathrm{P} \longrightarrow \wp(\mathrm{U} \times \wp([0,1])) \tag{5.22}$$

Second, we extend the eval° function, to return a list, explicitly declaring that all objects that would have a certain value are not present in it. Such a list is called *result set* and is represented as follows:

$$\begin{aligned} \mathrm{RESSET} &= \wp(\mathrm{U} \times \wp([0,1])) \times \wp([0,1]) \\ \mathrm{eval}^{\circledcirc}: \ \mathrm{product} \times \mathrm{P} &\longrightarrow \mathrm{RESSET} \end{aligned} \tag{5.23}$$

Note, that if all objects are present for a quality-property p, then

$$\mathrm{eval}^{\circledcirc}(pr, p) = (\mathrm{eval}^{\circ}(pr, p), \{\}) \tag{5.24}$$

Example: Result Set

An example for a result set may be:

$$resset_{example} = \Big(\big\{(a, \{1\}), (d, \{0.5\}), (f, \{0.7\})\big\}, \{0\}\Big) \tag{5.25}$$

This means that a has the value $\{1\}$, d has the value $\{0.5\}$, and f has the value $\{0.7\}$. All other objects than a, d, and f that exist in the product, but are not present in $resset_{example}$, have the value $\{0\}$.

To conveniently access the value of an object in such a result set, we define the val-function:

$$\begin{aligned} &\mathrm{val}: \ \mathrm{RESSET} \times \mathrm{U} \rightarrow \wp([0,1]) \\ &\mathrm{val}\Big(\big\{(o_1, v_1), \ldots, (o_n, v_n)\big\}, v_{missing}, object\Big) \mapsto \begin{cases} v_1 & \text{if } object = o_1 \\ \ldots & \\ v_n & \text{if } object = o_n \\ v_{missing} & \text{if } object \notin \{o_1, \ldots, o_n\} \end{cases} \end{aligned} \tag{5.26}$$

To access the set of all present objects we define:

$$\begin{aligned} &\text{objects}: \ \text{RESSET} \rightarrow \wp(\text{U}) \\ &\text{objects}\big(\{(o_1,\, v_1),\, \ldots,\, (o_n,\, v_n)\},\, v_{missing}\big) \mapsto \{o_1,\, \ldots,\, o_n\} \end{aligned} \tag{5.27}$$

In the example of Formula 5.25, the following queries can be run:

$$\begin{aligned} \text{val}(resset_{example},\, a) &= \{1\} \\ \text{val}(resset_{example},\, b) &= \{0\} \\ \text{val}(resset_{example},\, c) &= \{0\} \\ \text{val}(resset_{example},\, d) &= \{0.5\} \\ \text{val}(resset_{example},\, e) &= \{0\} \\ \text{val}(resset_{example},\, f) &= \{0.7\} \end{aligned} \tag{5.28}$$

Calculations on Incomplete Lists

In analogy to the °-operator defined to convert a function on $[0,1]$ to a function handling incomplete data, we now define the operator $^{\circledcirc}$ converting a function on $[0,1]$ to a function handling result sets. A function $f:\ [0,1]^n \rightarrow [0,1]$ is converted to $f^{\circledcirc}:\ \text{RESSET}^n \rightarrow \text{RESSET}$, as follows:

$$\begin{aligned} &f^{\circledcirc}(resset_1,\, \ldots,\, resset_n) \mapsto \\ &\quad\left(\left\{(o_r,\, v_r): \begin{array}{l} o_r \in \text{objects}(resset_1) \cup \cdots \cup \text{objects}(resset_n) \wedge \\ v_r = f^{\circ}\left(\text{val}(resset_1,\, o_r),\, \ldots,\, \text{val}(resset_n,\, o_r)\right) \end{array}\right\},\ f^{\circ}(v_{1_{missing}},\, \ldots,\, v_{n_{missing}})\right) \end{aligned} \tag{5.29}$$

Example

As an example we illustrate how a simple addition of real numbers takes place on result sets. The addition function is defined as follows:

$$\begin{aligned} add:\ &\mathfrak{R} \rightarrow \mathfrak{R} \\ add(x,y) &\mapsto x+y \end{aligned} \tag{5.30}$$

As input data, we have two result sets.

$$\begin{aligned} set_1 &= \left(\left\{\begin{array}{c} (o_1,\, \{0\}), \\ (o_2,\, \{3\}), \\ (o_4,\, \{2\}) \end{array}\right\},\, \{1\}\right) \\ set_2 &= \left(\left\{\begin{array}{c} (o_2,\, \{1\}), \\ (o_3,\, \{1\}), \\ (o_4,\, \{2\}) \end{array}\right\},\, \{0\}\right) \end{aligned} \tag{5.31}$$

We then apply $add^{\circledcirc}$:

$$
\begin{aligned}
& add^{\circledcirc}(set_1, set_2) \\
= & \left(\left\{ \begin{array}{l} (o_1, add^{\circ}(\{0\}, \{0\})), \\ (o_2, add^{\circ}(\{3\}, \{1\})), \\ (o_3, add^{\circ}(\{1\}, \{1\})), \\ (o_4, add^{\circ}(\{2\}, \{2\})), \end{array} \right\}, add^{\circ}(\{1\}, \{0\}) \right) \\
= & \left(\left\{ \begin{array}{l} (o_1, \{0\}), \\ (o_2, \{4\}), \\ (o_3, \{2\}), \\ (o_4, \{4\}) \end{array} \right\}, \{1\} \right)
\end{aligned}
\tag{5.32}
$$

Here we can see that the addition of two result sets yields a result set containing all objects of the input sets. For each object its two input values have been added.

Using this approach, measurement data of rule-based static code analysis tools can be aggregated like other measurement results. This works as described for divisions and restrictions. A special treatment is only necessary for impacts. The impact aggregation function Formula 5.6 is generally not calculable if there are unknown objects. However, the aggregation using ranges is calculable nonetheless: In the result set, either the objects with value 0 or with value 1 are present; consequently it is possible to either calculate the size of affected or of unaffected objects and since the overall size of the product is known, the ratio can be calculated.

5.4.4 Summary

First, we introduced the basic concept of conducting quality assessments with quality models and defined basic terms regarding measurement theory. Second, we introduced a general approach for operationalizing a quality model by defining concrete specifications how quality-properties can be measured and how measurement data can be aggregated for generating an overall quality statement. This approach is inspired by conducting automated quality assessments for source code, but it is generally applicable nonetheless.

From the third section onward, our focus is on automated tool-based quality assessments applied to source code. We explain typical constraints when applying real-world analysis tools and introduce a bottom-up assessment approach coping with these constraints. Subsequently, we address needs of practice by introducing approaches for handling incomplete measurement data and finding-based measurement data.

6 Quality Model for Software Products

Building a quality model conforming to the meta-model of Chapter 4 consists of three main steps: First, quality attributes are defined. Second, the classes, i.e., the product model, is defined. Third, based on the classes, concrete component properties and their impacts on the quality attributes are defined. The three types of elements created thereby are of different specificity: The quality attributes are mostly identical for all types of software products, while the component properties are always specific to a certain technology.

In Section 6.1, we introduce the parts of a quality model that can be defined generally for all types of software products. These are a high-level definition of artifacts of software products that are created for most software products and quality attributes. We define a hierarchy of quality attributes, which is created using the concept of activity-based quality models. This way, we achieve a clear decomposition of the quality attributes. Component properties are not defined in this section.

In Section 6.2, we present an operationalized quality model for Java source code. This model extends the general model of Section 6.1 by defining component properties referring to constructs of Java source code and by operationalizing them according to the quality assessment approach of Chapter 5.

6.1 General Classes and Quality Attributes

In Section 6.1.1, we define a general class model that is applicable to most software products. The defined classes are on a high level of abstraction and have to be extended for quality models referring to specific technologies. In the next subsections, we define a hierarchy of quality attributes. In literature a series of quality models defining quality attributes are proposed. However, as discussed in Section 3.1.1, the quality attributes defined by these models are criticized for being ambiguous, incomplete, and overlapping. Our approach to defining a quality attribute hierarchy overcoming this problem relies on the concept of activity-based quality models: Most qualities being important for stakeholders can be best expressed by activities conducted by these stakeholders with the software product. Thus, in Section 6.1.2, we first define a hierarchy of typical activities conducted with software products. For activities, clear decomposition criteria are available, which leads to a clear and unambiguous definition of the activities. Then, based on the hierarchy of activities, a hierarchy of quality attributes is derived in Section 6.1.3.

6.1.1 General Classes

The artifacts a software product consists of are defined as classes according to the meta-model introduced in Chapter 4. For the definition of the classes we partly rely on the V-Model XT (VMXT) [150] and IEEE-1074 [63]. The VMXT defines a comprehensive artifact model, describing all artifacts that are produced by the defined activities of their process model. However, all these artifacts have the character of documents created in the development process, e.g., the activity "preparing system architecture" creates a document "system architecture" describing the architecture of the system. This view on artifacts is not suitable for our quality model. If we were to define a quality-property of the class "system architecture", we would define characteristics of the document, instead of the architecture itself. Rather, we want to model that the system itself consists of an "architecture", interpreted as a certain view on the system. Therefore as a major concept in the class model, we define a class *system*, which has different *views* as parts. As one source for views, we used the architecture model for embedded systems of Broy et al. [18]. This defines three abstraction levels with differing views on the system. Additionally, we defined other views, based on the artifacts derived from IEEE-1074 and the VMXT. Figure 6.1 gives an overview of the hierarchical structure of the defined classes, which are defined in the following:

(Software) Product is the software artifact created by the development process.

(Software) System is a written program for a computer system.

Functional Usage View is a view on a software system, describing the behavior of the system at its interface to the environment [18].

Service Model is a way of describing the functionality of a software system by services. A service according to [18] is a set of possible interactions between the system and its environment.

Service Structure is a description of the system's functionality through a set of services which are hierarchically organized and may have cross-relations [18].

Service Interface is the description of the syntactic interface of the services [18] by defining their input and output ports and their data types.

Service Behavior is the description of the semantical interface of the services [18], e.g., by assumption/guarantee specifications or message-sequence-charts.

Use Case Model is a way of describing the functionality of a software system by use cases. A use case according to [21] is an informal description of possible interactions between the system and actors in the environment.

Actor is either a person or a system of the environment interacting with the system by its interface [21].

Scenario is an exemplary interaction between actors and the system, describing one possible set of exchanges messages [21].

Technical Usage View is a view on a software system describing the user interface that "enables information to be passed between a human user and [. . .] a computer system" [61].

Input-Output Model is a description of the devices and techniques used to transmit information between the system and its environment [42, 69].

GUI model is an example for an input output model usable for systems with a graphical user interface. It describes the graphical elements a graphical user interface consists of [42, 69].

Command model is an example of an input-output model for systems with a textual interface. It describes the syntax of a language for a textual command line interface [42, 69].

Dialogue Model is a description of the necessary interaction steps the user must carry out to achieve certain goals with the system [42, 69].

Logical Architecture View is a view on a software system describing the system by means of communicating logical components [18]. The focus of this view is not

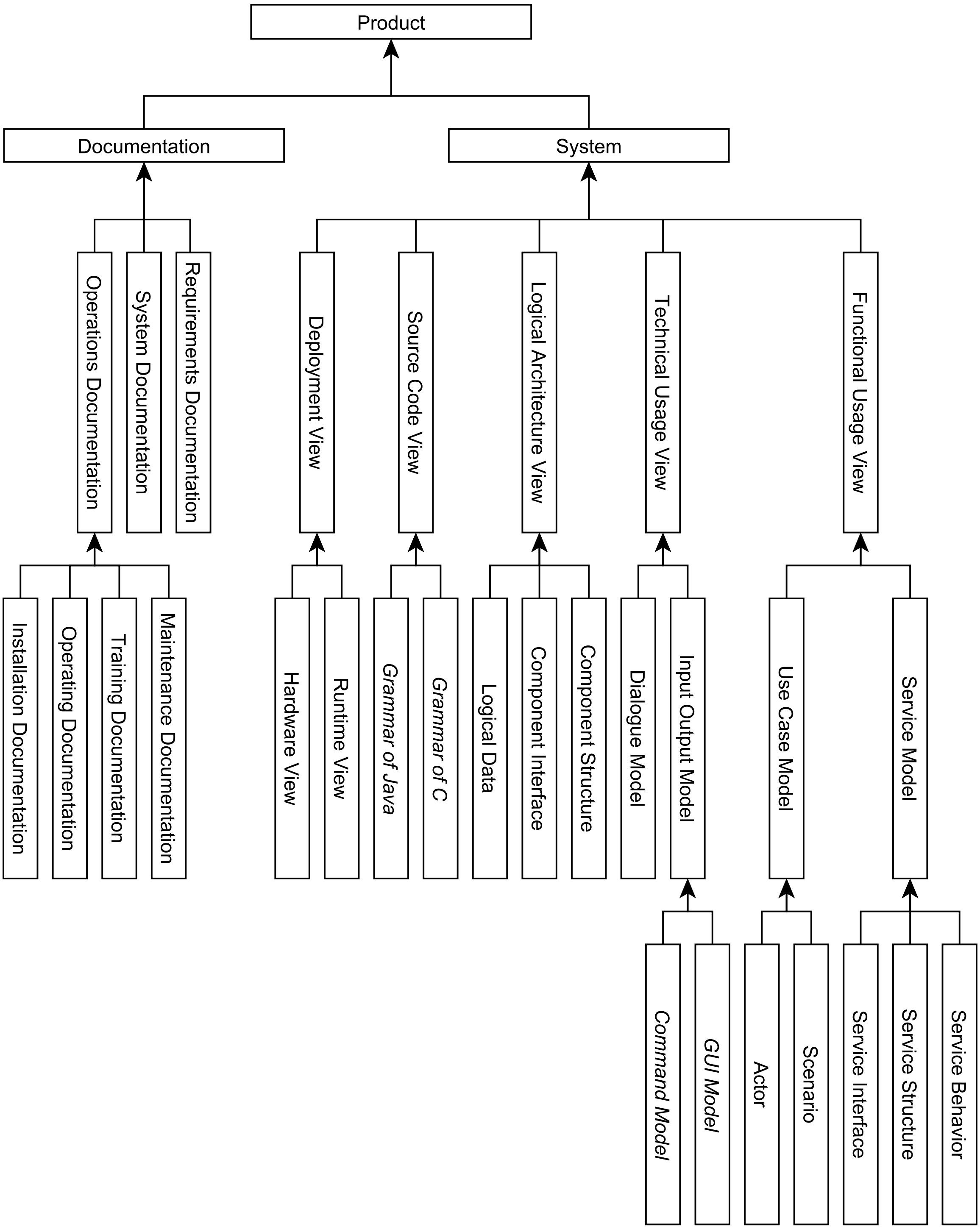


Figure 6.1: General Classes

the behavior of the system at its interface, but the decomposition of the system into logical components.

Component Structure is a description of all components, their syntactical interfaces and channels that interconnect the components [18].

Component Interface is the description of the syntactic interface of the components [18] by defining their input and output ports and their data types.

Component Behavior is the description of the behavior of the components by an operational description technique [18].

Source Code View is a view on a software system describing it by means of program code.

Grammar of Java is an example of a way of describing the constructs of the source code.

Grammar of C is an example of a way of describing the constructs of the source code.

Deployment View is a view on the software system on an abstraction level that is able to express time-constraints of the target platform the software will be running on [18], i.e., this view is able to express timing-constraints introduced by the operating system, buses, etc.

Runtime Model is a description of the behavior of the combined hardware/software, i.e., the behavioral characteristics of the hardware are combined with the behavioral characteristics of the logical architecture [18].

Hardware Model is a description of the hardware components on which the software will be running [18]. Behavioral properties of the hardware that have an influence on the software behavior are visible here.

Documentation is the entirety of all documents describing the software system.

Requirements Documentation is the document describing all "requirements posed on the system to be developed" [150].

System Documentation is the document describing the software system, i.e., it describes all views on the software systems defined above.

Operations Documentation is the document describing how typical actions are performed with/on the system by certain audiences.

Maintenance Documentation is the document describing "measures required in order to ensure and maintain the functional capability of the system" [150].

Training Documentation are all documents that are used in the training of personnel, such as "training manual, [...], outlines, text, exercises, case studies, visuals, and models" [63] (see also, "instructor documentation" in VMXT).

Operating Documentation , also called "user documentation" or "in-service documentation" (in VMXT), is "all data required by a user in order to operate the system properly and to respond adequately to problems" [150].

Installation Documentation is the document describing all actions to be taken to install the software system, e.g., "hardware and software configuration, [...], packaging, transport and storage" [150].

6.1.2 Activities

For the definition of activities that are conducted with or on a software system it is interesting to discuss the *generality* of the activities. One extreme case is those activities that are the quite similar for all software systems, for instance, maintenance. Maintenance can be defined as the sub-activities of analyzing the change-request, planning the change, modifying the product, etc. Another extreme is the activities that are individual to each software system, such as the activity of conducting a cash transfer in an online-banking system. For the description of quality, both extremes are relevant. For instance, think of the quality attribute *functional suitability* of ISO 25010. To assess it for the online-banking system, you have to know that cash transfers are an activity the system must support.

For deriving quality attributes, we should consider all activities that are general enough to be of interest, e.g., from a typical quality model for software quality it is expected that it defines maintainability as a quality attribute.

In this section, we present a collection of general activities being relevant for most software systems. As a source for such activities we use lifecycle models of software that are available in the literature. As sources for defining these activities, we mostly use the IEEE-1074 [63] (*Standard for Developing a Software Project Life Cycle Process*) and the process model *V-Modell XT*(VMXT) [150]. Furthermore, we distill additional activities from the definition of the quality attributes of the ISO 25010 [65]. The IEEE-1074 is kind of a meta-standard defining a process to create a software lifecycle process. However, besides that meta-process, in its annex it gives a collection of normative activities covering software lifecycle processes in general. Thus, these normative activities are a comprehensive collection suited to being used in our activity model. The VMXT is a process model for planning and executing projects. It is mandatory for all software development projects initiated by the Federal Republic of Germany. Its scope is wider than that of IEEE-1074, focusing to a large amount on project management, contracting, and contractor/supplier relationships. Thus, it is a good complement to IEEE-1074 for defining activities in software development.

The two standards mentioned above are especially useful in defining "high-level" activities of the software lifecycle. When going into more detail, e.g., for defining fine-grained sub-activities of software-maintenance, we refer to more specialized standards, such as IEEE-1219 [62] (*Standard for Software Maintenance*).

Figure 6.2 gives an overview of the hierarchical structure of the defined activities. The order of the activities does not imply the sequence of their execution. However, we ordered them into a typical sequence, as given by a general v-model, so, for example, all testing and evaluation activities come after the design and implementation activities, although they may be performed in an interleaving manner. In the following, the activities of Figure 6.2 are defined:

Lifecycle is the entirety of all activities conducted in software development from the first idea to the realization and disposal of a software product.

Project Management is the entirety of all accompanying activities "that initiate, monitor, and control a software project throughout its life cycle." [63]

Project Initiation is an example of a project management activity. It is usually the first activity of a software development project "that creates and updates the infrastructure of a software development [...] project." [63]

Managing Risks is an example of a project management activity. It is the entirety of all activities that "iteratively analyze and mitigate business, technical, managerial, economic, safety, schedule, and security risks." [63]

Train New Developer on Maintenance is an example of a project management activity. It is the activity of training a new employee in conducting maintenance activities. This activity is an example of de-

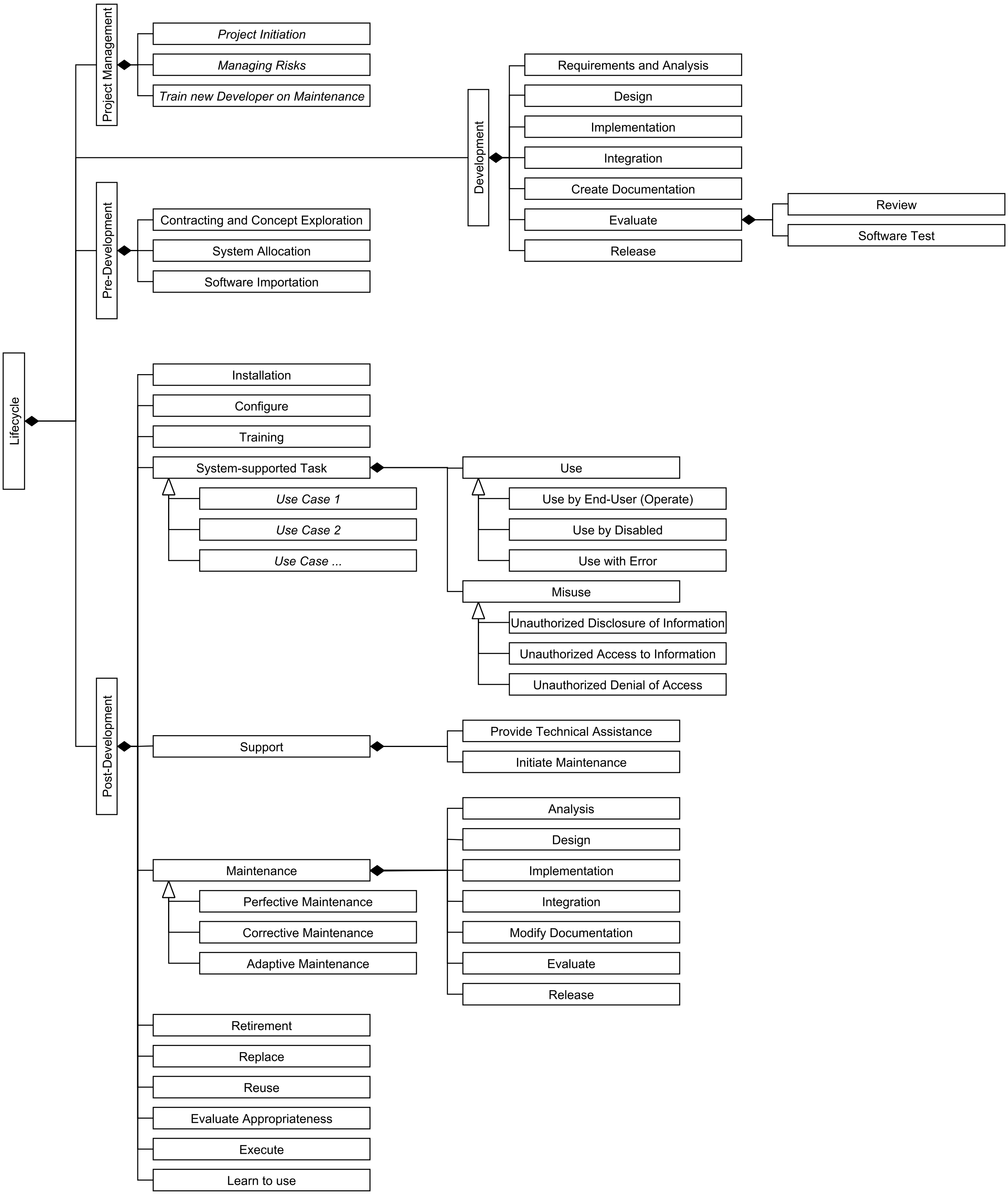


Figure 6.2: Activities

scribing a non-classical quality attribute. Easily training new developers on maintenance lately became important because maintenance tasks are often outsourced to suppliers and subject to fluctuations in employees.

Pre-Development is the entirety of all activities that are conducted before the development of the software can begin. It includes the definition of the project goal and the contracting [63].

Contracting and Concept Exploration are all activities concerned with the formal setup of the project and the definition of high-level project goals.

System Allocation, according to IEEE-1074, describes the decision about the scope of the software, in terms of taking the decision which parts of the defined project goals may be realized by hardware, software or people.

Software Importation, according to IEEE-1074, is the entirety of all activities to define the software that can be reused, to determine the method of importation and to "import the software including documentation" [63].

Development is the entirety of all activities conducted to actually construct the software product.

Requirements and Analysis, according to the VMXT, are the activities to define "user requirements based on a project proposal (prestudy) and the contract." The IEEE-1074 defines a corresponding activity "software requirements".

Design, according to IEEE-1074, is the entirety of all activities used for developing an architectural design, constituted of components defining their structure and interfaces. At a detailed design level, data structures and algorithms are selected.

Implementation is defined by the IEEE-1074 as a transformation of the design representation of the software product into a programming language realization.

Integration is the activity of collecting all artifacts created in the implementation and connecting them to the actual software product.

Evaluate, according to IEEE-1074, is the entirety of all activities "that are designed to uncover defects in the product or processes that are used to develop the product".

Review is the entirety of all activities that manually inspect an artifact with the goal of finding deficiencies. They can be applied to all artifacts created in the development activity, like requirements documents, design specifications, and the implementation [63].

Software Test is the entirety of all activities to prepare and conduct a test of the software. According to IEEE-1074, tests can be conducted at various levels, such as unit/component/system tests and using various techniques, such as static/dynamic.

Release is the entirety of all activities for "building, naming, packaging, and release of a particular version of a software product, including its associated release notes, user documentation" [63].

Post-Development is the entirety of all activities that are performed after the actual development of the software.

Installation is the entirety of all activities "consisting of transportation and installation of a software system from the development environment to the target environment" [63].

Configure is the activity of adapting the software product to certain circumstances by means of using built-in functionality of the software product.

Training is the entirety of all activities to implement the actual training of personnel that install, operate, and maintain the software, such as the "provision of all necessary materials, the arrangement of the locations and facilities for training, and the delivery of the training" [63].

System-Supported Task is an activity the user conducts by using the software product. In business information systems, such system-supported tasks are the business processes that are system-supported. Usually they are described by use cases in requirements specifications.

Use is the activity of using the product. Using the product includes interacting with the system and reading the documentation.

Use by End User (Operate) is a special type of usage where the actor is an end user. Thus, this activity is also called "operate".

Use by Disabled is a special type of usage where the actor is a disabled person.

Use with Error is a special type of usage where the user performs the interaction accidently or with an error.

Misuse is a special type of usage, where the user is not authorized to interact with the system. Unauthorized users deliberately interacting with the system usually have a hostile intent and are called attackers; this activity is also called "attack".

Unauthorized Disclosure of Information is the activity of disclosing data without authorization.

Unauthorized Access to Information is the activity of accessing (including modifying) information without authorization.

Unauthorized Denial of Access is the activity of denying access to authorized users without authorization.

Use Case 1, 2, ... are use cases that are individual to each software system. Use cases define how the system is used by the user in order to conduct a business task.

Support is the entirety of all activities "providing technical assistance, consulting with the user, recording user support requests" [63] and triggering maintenance activities.

Provide Technical Assistance and Consulting is the entirety of all activities "responding to the user's technical questions or problems" [63].

Initiate Maintenance, according to IEEE-1074 and IEEE-1219, is the entirety of all activities to collect change requests from users, to formulate modification requests, and to trigger the maintenance process.

Maintenance is the "modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment" [62].

Analysis is the entirety of all activities to "study the feasibility and scope of the modification and to devise a preliminary plan for design, implementation, test, and delivery" [62].

Design is the entirety of all activities to design the modification of the system using "all current system and project documentation, existing software and databases, and the output of the analysis phase" [62]

Implementation is the entirety of all activities to perform the modifications on the system by using "the results of the design phase, the current source code, and project and system documentation" [62].

Integration is the analogy to the activity "integration" in the development.

Modify Documentation is the analogy to the activity "create documentation" in the development.

Evaluate is the analogy to the activity "evaluate" in the development, however evaluation in maintenance especially includes "regression testing [...] to validate that the modified code does not introduce faults that did not exist prior to the maintenance activity" [62].

Release is the analogy to "release" in development. Additionally, it includes archiving the replaced system [62].

Perfective Maintenance is a special type of maintenance, where the goal of the activity is to enhance the product [62].

Corrective Maintenance is a special type of maintenance, where the goal of the activity is to correct faults [62].

Adaptive Maintenance is a special type of maintenance, where the goal of the activity is to adapt the product to a new environment [62]. This is also called "to port" the product or "to adapt" the product.

Retirement is the entirety of all activities for the "removal of an existing system from its active support or use either by ceasing its operation or support or by replacing it with a new system or an upgraded version of the existing system" [63].

Replace is the activity of replacing an existing product with the newly developed one.

Reuse is the activity of using the product as part of another software.

Evaluate Appropriateness is the activity of evaluating whether the product is suited for the needs of a certain user.

Execute is the activity of executing the software on the given hardware.

Learn to Use is the activity of learning how to use the product.

6.1.3 Quality Attributes

The definition of the quality attributes is based on the concept of activity-based quality models. Such quality models define quality attributes by referring to activities that are conducted with or on the system. For instance, the classical quality attribute *maintainability* is described as the efficiency and effectiveness of conducting the *maintenance* activity. This definition of maintainability is also in concordance with the definition in ISO 25010 [65], stating *the degree of effectiveness and efficiency with which the product can be modified*. The ISO 25010 refers to the activity of modifying the product, while we explicitly talk about the maintenance activity.

The benefit of reasoning about activities is that they provide a clear decomposition criterion: Activities may be decomposed into sub-activities. For instance, maintaining a system means conducting the following sub-activities: analyzing the change request and the existing system, modifying the system, and verifying the modified system. This leads to the introduction of analyzability, modifiability, and verifiability as sub-quality attributes of maintainability.

As a starting point for the development of our quality model, we use the quality attributes of ISO 25010 [65]. We redefine them according to the activities defined in Section 6.1.2, by removing and adding new quality attributes where necessary.

The main part of the developed quality model consists of a tree of quality attributes. This tree contains all quality attributes constituting the quality of a system. They were defined so as to be as non-overlapping as possible. Figure 6.3 shows the hierarchy of these quality attributes.

Besides the main tree of quality attributes, there is a list of auxiliary quality attributes, which are overlapping with the main quality attributes. Each of these quality attributes includes multiples of the main quality attributes or of parts of them. For instance, the auxiliary quality attribute portability is a combination of adaptability and releasability, whereby adaptability is a special type of maintainability.

A third kind of quality attributes are orthogonal to other kinds of quality attributes. They are provided as a separate list.

Figure 6.3 shows the quality attribute hierarchy. The quality attributes are defined thereafter.

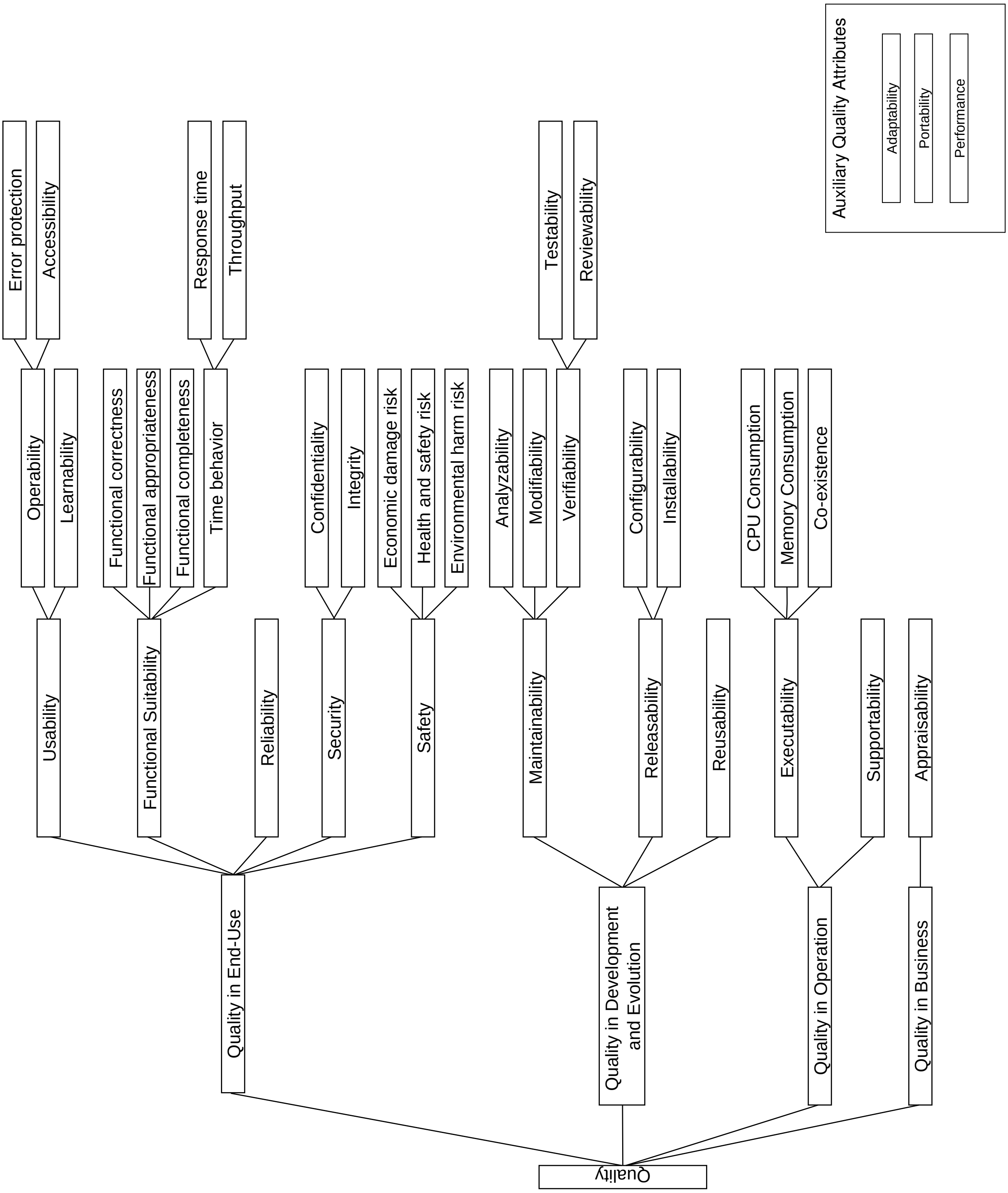


Figure 6.3: Quality Attribute Hierarchy

Main Quality Attributes

Quality. The degree to which the system satisfies the requirements of all stakeholders.

Quality in End-Use. The degree to which the system satisfies the requirements of the end user.

Usability. The degree to which the system enables effective use by the end user. Using the system includes operating (i.e., interacting with it) and reading/understanding the documentation.

Operability. The degree to which the system enables effective operation by the end user. Operation by the end user includes providing input to the system and perceiving and understanding the output of the system.

Error protection. The degree to which the system prevents the end user from wrong and/or accidental input to the system.

Accessibility. The degree to which the system enables end users with disabilities operating the system efficiently and effectively.

Learnability. The degree to which the documentation of the system is suited to efficiently and effectively instructing the end user in operating the system.

Functional Suitability. The degree to which the system provides functionality that supports the tasks of the end user.

Functional Correctness. The degree to which the system provides the correct results with the required degree of precision.

Functional Appropriateness. The degree to which the functionality of the system supports the tasks of the end user.

Functional Completeness. The degree to which the tasks of the end user are covered by the functionality of the system.

Time Behavior. The degree to which the system satisfies required response times and throughput rates.

Response Time. The degree to which the system satisfies required response times.

Throughput. The degree to which the system satisfies required throughput rates.

Reliability. The probability of the system to be functionally correct (see *functional correctness*) at any time.

Security. The degree to which the system prevents unauthorized actors from: (1) reading or modifying data of the system and (2) preventing authorized actors from using the system.

Confidentiality. The degree to which information and data are protected from unauthorized disclosure.

Integrity. The degree to which the system prevents unauthorized reading or modifying of data.

Safety. "[T]he degree to which a product or system does not, under specified conditions, lead to a state in which human life, health, property, or the environment is endangered" [65].

Economic Damage Risk. "[T]he degree of expected impact of harm to commercial property, operations or reputation in the intended contexts of use" [65].

Health and Safety Risk. "[T]he degree of expected impact of harm to people in the intended contexts of use" [65].

Environmental Harm Risk. "[T]he degree of expected impact of harm to property or the environment in the intended contexts of use" [65].

Quality in Development and Evolution. The degree to which the system satisfies the requirements of the stakeholders concerned with tasks regarding the development and evolution of the system. The evolution of a system includes maintaining and releasing it.

Maintainability. The degree to which the system can be maintained efficiently and effectively. Maintaining the system means the modification of the system to correct faults, to improve it to prevent future faults, or to adapt the product to satisfy changed requirements.

Analyzability. The degree to which the systems enables: (1) the study of the feasibility and scope of a requested modification and (2) the devising of a preliminary plan for design, implementation, test, and delivery.

Modifiability. The degree to which the systems can be modified by using the results of the design phase, the current source code, and project and system documentation.

Verifiability. The degree to which the system enables the test for satisfaction of the changed requirements.

Testability. The degree to which the system enables to conduct software tests to assess the satisfaction of requirements.

Reviewability. The degree to which the system enables to conduct reviews of it.

Releasability. The degree to which the system can be efficiently and effectively released to customers. Releasing means building, naming, packaging, releasing a particular version of the system, installing, and making it operational for the customers.

Configurability. The degree to which the system can be efficiently and effectively adapted to certain circumstances by means of using the built-in functionality of the system; i.e., without changing the system itself.

Installability. The degree to which the system can be efficiently and effectively installed and/or uninstalled in a specified environment.

Reusability. The degree to which the system can be efficiently and effectively used as part of another software.

Quality in Operation. The degree to which the system satisfies the requirements of the stakeholders concerned with operating the system. Operating includes operation of the hardware, and providing support to end users.

Executability. The efficiency with which the system can be executed on the target hardware.

CPU Consumption. The efficiency with which the system uses the computing resources of the CPU.

Memory Consumption. The efficiency with which the system uses the memory of the hardware.

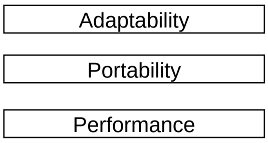
Co-existence. The degree to which the system can co-exist with other independent systems in a common environment, sharing common resources without any detrimental impacts.

Supportability. The degree to which the system enables providing technical assistance, consulting with the user, recording user support requests, and triggering maintenance activities.

Quality in Business. The degree to which the system satisfies the requirements of the stakeholders concerned with acquiring software.

Appraisability. The degree to which acquisitioners can efficiently and effectively assess whether the system satisfies their requirements.

Auxiliary Quality Attributes

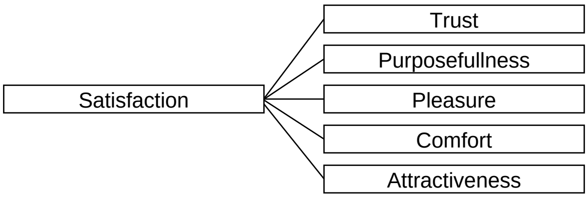


Adaptability. A special type of maintenance, with the goal of adapting the system to satisfy changed requirements. Two other special types of maintenance are corrective and preventive maintenance.

Portability. The degree to which the system can be efficiently and effectively transferred from one hardware, software or other operational or usage environment to another. Transferring the system means adapting it, and releasing it. Thus, *portability* is a combination of *adaptability* and *releasability*.

Performance. Subsumes the *time behavior*, *CPU consumption*, and *memory consumption*.

Orthogonal Quality Attributes



Satisfaction. The degree to which the system makes the end user feel satisfied by using it.

Purposefulness. The degree to which the end user "is satisfied with their perceived achievement of pragmatic goals, including acceptable perceived results of use and consequences of use" [65].

Trust. The degree to which the end user "is satisfied that the product will behave as intended" [65].

Pleasure. The degree to which the "end user obtains pleasure from fulfilling their personal needs" [65].

Attractiveness. The degree to which the end user considers the product to be attractive.

6.2 A Quality Model for Java Source Code

In this section, we describe a quality model for Java source code. First, we describe how it was developed based on the "base model" of the Quamoco project. Second, we explain its structure in detail.

6.2.1 Transformation of the Quamoco Model

The quality model for the Java source code is an adaptation of the *Quamoco* base model [126]. The Quamoco project developed a meta-model for quality models that has more degrees of freedom than our model. In the following, we introduce the main elements of it and our approach to converting it. Figure 6.4 shows the meta-model as an UML class diagram. The main concept of this meta-model is a *Factor*. A factor is the equivalent of a quality-property in our model. The concept of an *Entity* is the equivalent of our concept of a class. Quamoco differentiates between *Product Factors*, being equivalent to our component properties and *Quality Aspects*, being equivalent to our quality attributes. The main difference to our model is that in Quamoco there is only one relation between factors, called *Refinement*. This relation can be used arbitrarily by the modeler, i.e., it is not constrained by the entities and their relations. Furthermore, in Quamoco the normalizations/evaluations/aggregations are arbitrarily specified by the modeler by *Evaluations*, which allow the use of a functional programming language.

The Quamoco model was automatically transformed to a model (partially) conforming to our meta-model. The transformation of elements and relations was performed as shown in Table 6.5. For deciding whether a *Refinement* must be converted to a restriction or division, a heuristic was used: If the entity of two refining factors is the same, then a division is used; in all other cases a restriction is used.

After the automatic transformation, the resulting quality model did not fully conform to our meta-model. Hence, in the second step the entire model was manually reviewed and adapted to the concepts of our meta-model. In particular, all evaluation/aggregation specifications had to be redone, because our concept of operationalization is profoundly different to that of Quamoco. In Quamoco all measurement values are aggregated to the product level first and then aggregated alongside the

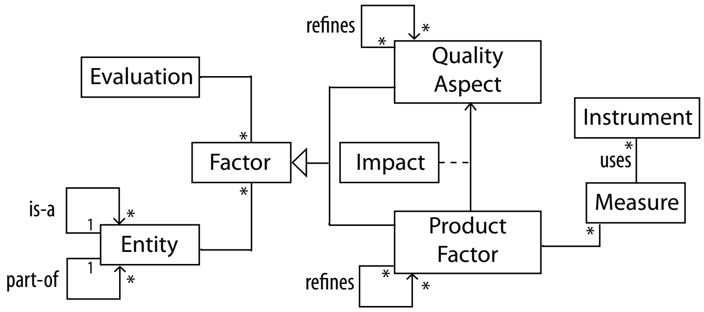


Figure 6.4: Quamoco Meta-Model

Quamoco Model		Our Model
Entity	→	Class
Factor	→	Quality-Property
Refinement	→	Restriction *OR* Division
Impact	→	Impact
Measure *AND* Instrument	→	Measure

Figure 6.5: Transformation of Model Elements

structure of the factor hierarchy, whereas in our model the aggregation steps take place individually for each object and the aggregation to product level is done at the impacts.

The definition of the evaluation and aggregation specifications in particular includes defining threshold values for fuzzification functions, defining threshold values for distribution functions at impacts, and defining ranks/weights for weighted-sum aggregations in the quality attribute hierarchy. In the following, for all model elements we describe how they were filled-in by either expert opinion or benchmarking data.

Fuzzification of Measurement Values (see Section 5.2.2)
The threshold values for the mapping functions for measurement values are derived from experts. First, the answers of the different experts are collected. Then, the agreement of the different experts is determined by calculating the variance of the values. As a threshold in the quality model, the mean of the values is used.

Weighting of Impacts (see Section 5.2.5)
For each impact in the quality model, a way of weighting has to be chosen. For all impacts having complete data available, we use the *mean* function, calculating the mean of the values of all objects. For impacts relying on data based on findings, ranges are used. Thus, for each impact an appropriate range class has to be chosen. As a predefined range-class we automatically choose the *smallest* class, i.e., the deepest possible class in the composition hierarchy. For instance, if a range-class for *expression* has to be chosen, there are three possibilities: *file*, *class*, and *method*. In this case *method* is chosen, because it is part-of the others. These automatically-determined range-classes are reviewed manually and changed if necessary.

Ranking of Aggregations of Impact Values (see Section 5.2.5)
Quality attributes impacted by component properties use ranking aggregations to aggregate the values of multiple impacts. The ranks of the single impacts are chosen by expert opinion. As a result of the high number of impacts (370) in the Java quality model, the ranks were not chosen independently by multiple experts and then checked for correspondents, but instead were set by an experts' discussion. As mentioned in Section 5.2.5, rankings are well suited to being discussed and set in group discussions. As a support, the following guideline for assigning a rank to an impact was defined: Consider a component property has an impact on a quality attribute; then the following ranks are given:

1. *Rank 1:* The impact receives the rank of 1 if the quality-property has an outstanding influence on the quality attribute. For instance, $[\mathit{Cloning}|\mathit{Source\ code}]$ is known as having a major impact on modifying the source code, because each modified part of the source code must be checked for clones that may have to be changed too.

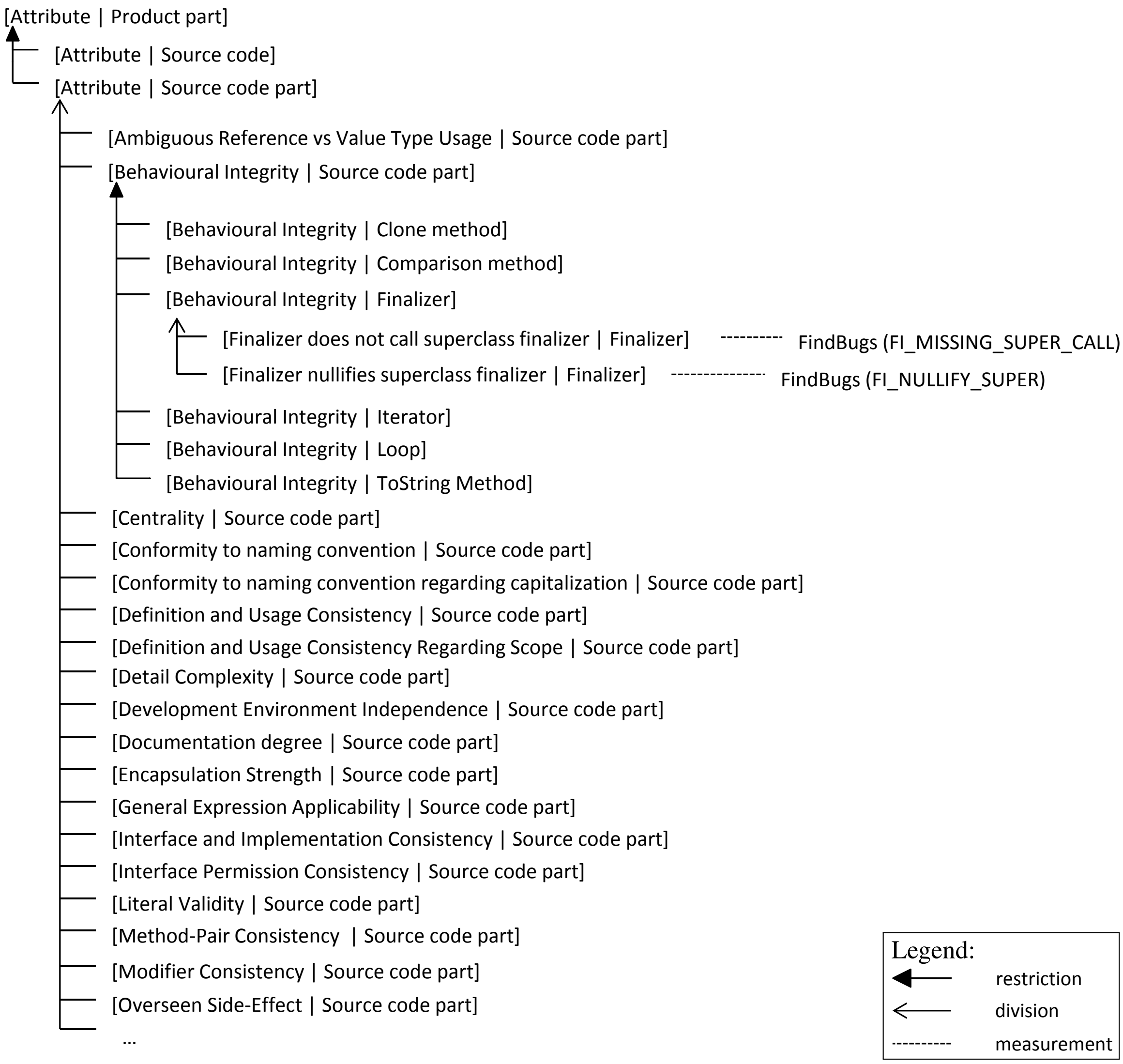

[Attribute | Product part]
[Attribute | Source code]
[Attribute | Source code part]
[Ambiguous Reference vs Value Type Usage | Source code part]
[Behavioural Integrity | Source code part]
[Behavioural Integrity | Clone method]
[Behavioural Integrity | Comparison method]
[Behavioural Integrity | Finalizer]
[Finalizer does not call superclass finalizer | Finalizer]
FindBugs (FI_MISSING_SUPER_CALL)
[Finalizer nullifies superclass finalizer | Finalizer]
FindBugs (FI_NULLIFY_SUPER)
[Behavioural Integrity | Iterator]
[Behavioural Integrity | Loop]
[Behavioural Integrity | ToString Method]
[Centrality | Source code part]
[Conformity to naming convention | Source code part]
[Conformity to naming convention regarding capitalization | Source code part]
[Definition and Usage Consistency | Source code part]
[Definition and Usage Consistency Regarding Scope | Source code part]
[Detail Complexity | Source code part]
[Development Environment Independence | Source code part]
[Documentation degree | Source code part]
[Encapsulation Strength | Source code part]
[General Expression Applicability | Source code part]
[Interface and Implementation Consistency | Source code part]
[Interface Permission Consistency | Source code part]
[Literal Validity | Source code part]
[Method-Pair Consistency | Source code part]
[Modifier Consistency | Source code part]
[Overseen Side-Effect | Source code part]
...
Legend:
restriction
division
measurement

Figure 6.6: Structure of the Quality-Property Hierarchy in the Java Model
(Excerpt of the full model)

2. *Rank 2:* The impact receives the rank of 2 if there is a clear and direct connection from the quality-property to the quality attribute. For instance, the impact of [*Synchronization integrity*|*Class*] on reliability got a rank of 2, because wrong synchronization may lead to deadlocks. Obviously, a deadlock results in a crash, which is clearly related to reliability.
3. *Rank 3:* The impact receives the rank of 3 if there is only an indirect connection from the quality-property to the quality attribute. The same quality-property as in the example above – [*Synchronization integrity*|*Class*] – receives the rank 3 for its impact on analyzability, because classes handling synchronization in a rigorous manner are usually easier to understand. Here, the influence is considered only indirect and minor, because bad synchronization is not as directly related to analyzability as, for instance, meaningful identifiers.
4. *Rank 4:* An impact receives the rank of 4 if the influence if considered minimal, although worth modeling. For instance, the impact of [*Synchronization overhead*|*Class*], describing that a method is synchronized unnecessarily, on *time behavior* got rank of 4, because entering and exiting a synchronized block only consumes very little time in comparison to entering and exiting non-synchronized methods.

6.2.2 Structure of the Java Model

The model contains 10 quality attributes, which are impacted by component properties. The impacts in the quality model always target leaf quality attributes. The model contains 370 impacts, 566 component properties, of which 378 are leaves. Figure 6.6 shows the structure of the component property hierarchy. The root quality-property is [*Attribute*|*Product part*], which is restricted by two quality-properties which are specific for source code. The quality-property [*Attribute*|*Source code part*] serves as a root quality-property for all quality-properties referring to constructs of the source code. It is divided into several quality-properties that describe general attributes of source code, e.g., *Behavioral integrity*, defined as follows: “An object’s behavioral integrity is present if it behaves according to the semantic assumptions associated with it”. Each quality-property on this level of the hierarchy is restricted by quality-properties with specific classes, such as *Clone method*, *Comparison method*, or *Finalizer*. These quality-properties are already concrete enough to be measured by their direct children. The quality-property [*Behavioral integrity*|*Finalizer*], for instance, has two sub-quality-properties [*Finalizer does not call superclass finalzer*|*Finalizer*] and [*Finalzer nullifies superclass finalizer*|*Finalizer*], which are directly measured by FindBugs rules.

7 Tool Support

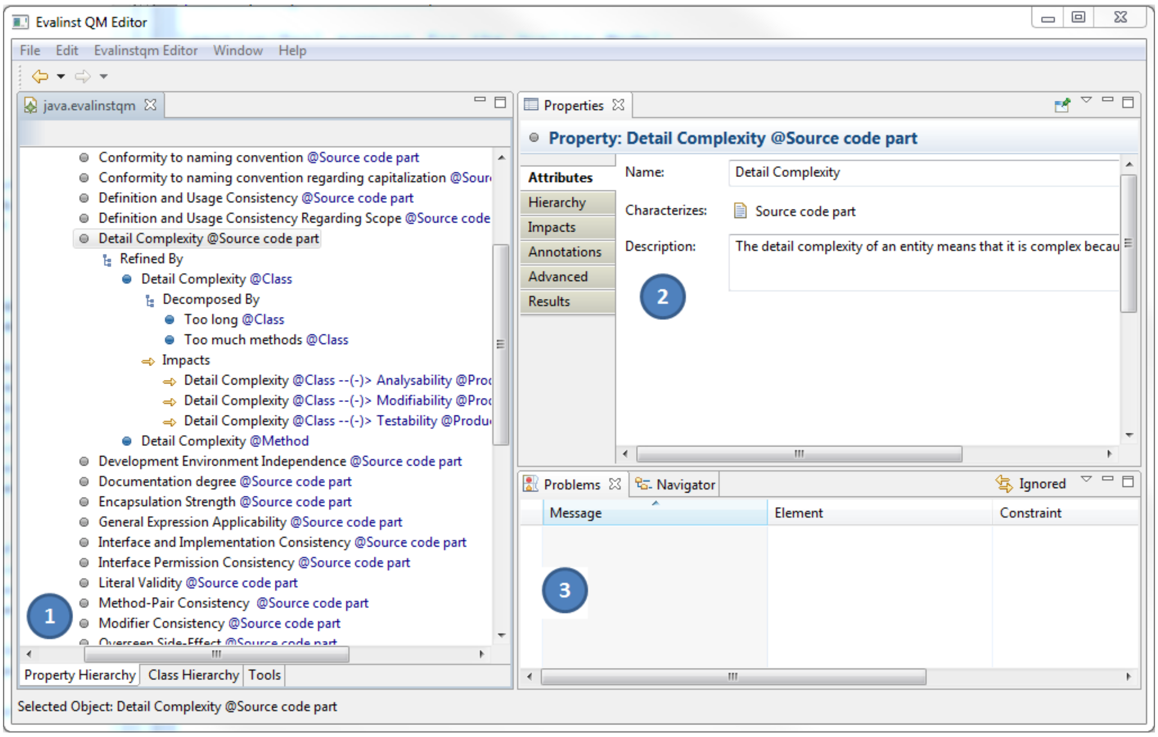
In this chapter, we describe the two parts of the tool support. First, the *quality model editor* is introduced, which is used to create and edit quality models conforming to the given meta-model. Second, we introduce the *quality assessment framework*, which conducts quality assessments based on a given quality model.

7.1 Quality Model Editor

Real quality models usually grow very large and consist of several hundred model elements. Thus, tool support is needed to build quality models conforming to the given meta-model. The quality model editor developed for this thesis is based on the editor developed in the Quamoco project [27]. Due to the different meta-model and the completely different aggregation specifications, the editor is a substantial modification of the Quamoco editor. Figure 7.1 shows the main user interface of it. It consists of the following elements:

1. *Hierarchical Views:* The content of the quality model is represented in the form of several views onto it:
   a) *Property Hierarchy View* shows all quality-properties in the hierarchy induced by the relations *restriction* and *division*.
   b) *Class Hierarchy View* shows all classes in the hierarchy induced by the relations *generalization* and *composition*.
   c) *Tool View* allows the tools used for measuring quality-properties to be modeled. They are needed for the connection to the assessment toolkit.
2. *Content Editor:* If an element is selected in a *hierarchy view* then its attributes can be edited in this view.
3. *Problems View:* The editor continuously checks conformity of the built model to the meta-model. All violations are shown in this view.

In the property hierarchy, the directed acyclic graph formed by the restriction/division relations is represented as a tree. Obviously, there are multiple ways of transforming this graph into a tree. In order to ensure an unambiguous representation of the quality-property tree, we do not allow two children of a node to be in a restriction relation. Figure 7.4 shows how a restriction-hierarchy is represented in the editor. Figure 7.4a shows the restriction relation of the actual quality model. Figure 7.4b shows the most straightforward representation of the model as a tree; only transitively induced restriction relations are omitted. Figure 7.4c shows that, for instance, the quality-property $[a|c_3]$ can be omitted, so that $[a|c_4]$ directly restricts $[a|c_1]$. Not allowed is the tree of Figure 7.4d,



Conformity to naming convention @Source code part
Conformity to naming convention regarding capitalization @Sour
Definition and Usage Consistency @Source code part
Definition and Usage Consistency Regarding Scope @Source code
Detail Complexity @Source code part
  Refined By
    Detail Complexity @Class
      Decomposed By
        Too long @Class
        Too much methods @Class
      Impacts
        Detail Complexity @Class --(-)> Analysability @Proc
        Detail Complexity @Class --(-)> Modifiability @Proc
        Detail Complexity @Class --(-)> Testability @Produ
    Detail Complexity @Method
Development Environment Independence @Source code part
Documentation degree @Source code part
Encapsulation Strength @Source code part
General Expression Applicability @Source code part
Interface and Implementation Consistency @Source code part
Interface Permission Consistency @Source code part
Literal Validity @Source code part
Method-Pair Consistency @Source code part
Modifier Consistency @Source code part
Overseen Side-Effect @Source code part

Property: Detail Complexity @Source code part

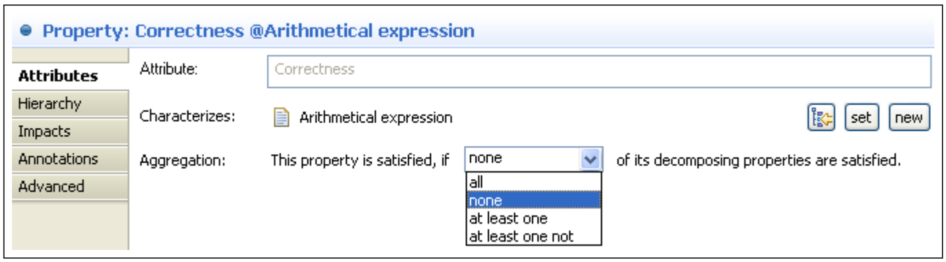
Name: Detail Complexity

Characterizes: Source code part

Description: The detail complexity of an entity means that it is complex becau

Selected Object: Detail Complexity @Source code part

Figure 7.1: Screenshot: Main View of the Quality Model Editor



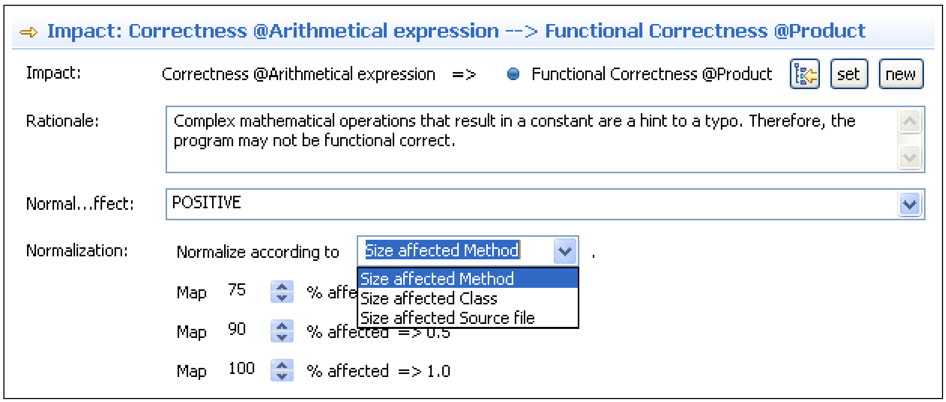
Property: Correctness @Arithmetical expression

Attribute: Correctness

Characterizes: Arithmetical expression

Aggregation: This property is satisfied, if none of its decomposing properties are satisfied.

Figure 7.2: Screenshot: Editing a Quality-Property that is Divided into Other Properties



Impact: Correctness @Arithmetical expression --> Functional Correctness @Product

Impact: Correctness @Arithmetical expression => Functional Correctness @Product

Rationale: Complex mathematical operations that result in a constant are a hint to a typo. Therefore, the program may not be functional correct.

Normal...ffect: POSITIVE

Normalization: Normalize according to Size affected Method .

Map 75 % affe

Map 90 % affected => 0.5

Map 100 % affected => 1.0

Figure 7.3: Screenshot: Editing an Impact

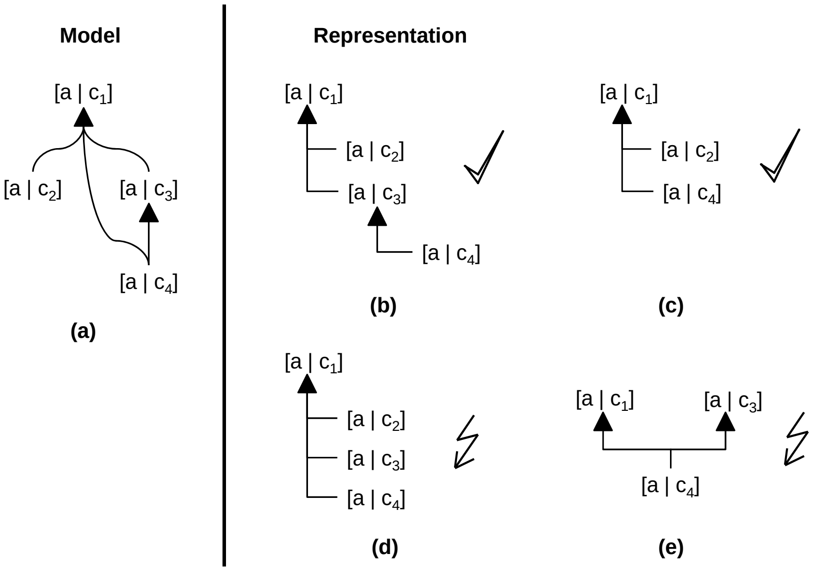


Figure 7.4: Representation of Restrictions in the Editor

where both $[a|c_3]$ and $[a|c_4]$ are direct children of $[a|c_1]$. Obviously the construct in Figure 7.4e is also not allowed, because it contradicts the definition of the restriction relation.

User Support by the Editor

In order to facilitate the creation of valid quality models, the editor supports the user in the following ways. When a new quality-property is added to this hierarchy, the user has to decide if it is restricting or dividing its parent-property. The editor automatically sets the name or the class of the quality-property correctly, i.e., a restricting quality-property must have the same name as its parent-property (Section 4.4.1) and a dividing quality-property must have the same class as its parent property (Section 4.4.2). Furthermore, when changing the name or class of a quality-property, the editor only allows the creation of valid models:

1. That means if a quality-property is restricted by another quality-property, its name cannot be changed (in the screenshot in Figure 7.2 the field name is disabled) and if the name of a quality-property is changed, then the names of all restricting quality-properties are changed automatically.
2. If the class of a restricting quality-property is changed, only subclasses of the class of its parent quality-property are shown for selection (Section 4.4.1). Furthermore, the editor checks the constraint for inheritance (Section 4.4.3) and shows a warning if a quality-property is not correctly inherited alongside the restriction hierarchy.

For the automated quality assessment, aggregation functions must be specified in the quality model. For instance, if a quality-property is divided into other quality-properties, it is necessary to specify an aggregation function, as explained in Section 5.2.4. The four aggregation functions that were identified as commonly used are integrated into the editor. They can be specified using sentence patterns as shown in the screenshot in Figure 7.2.

For each impact in the model, it must be specified how it is weighted. The concept of ranges (see Section 5.2.5) is fully integrated into the editor. Depending on the domain of the quality-property under consideration, a list of all possible ranges is determined. This takes the composition relation of the class and the available measures of the tooling into consideration. Furthermore, the linear distribution function and its thresholds can be edited in a table. Figure 7.3 shows a screenshot of the respective form, whereby the impact $[\mathit{Correctness}|\mathit{Arithmetical\ expression}]\xrightarrow{\oplus}[\mathit{Functional\ correctness}|\mathit{Product}]$ is being edited. The user can choose between three ranges: *method*, *class*, and *source file*. For the linear distribution function, the user can enter the threshold values. In the example, the following mapping was entered: 75% affected $\mapsto 0.0$, 90% affected $\mapsto 0.5$, and 100% affected $\mapsto 1.0$.

Checking of Constraints in the Editor

The editor checks meta-model constraints and reports its violations. Subsequently, we describe the checks firing most frequently.

All constraints mentioned in the scenarios above are also checked as constraints. Although the wizards of the editor prevent the creation of invalid models in a straightforward manner (e.g. it prevents the selection of an invalid range-class), there are other ways to invalidate the model. For instance, instead of selecting an invalid range-class, the user can change the container of a class already used as range-class, so that the class is no longer valid as class-range. To detect invalid models constructed this way, all constraints used by the various wizards are checked independently and violations are reported.

In addition to the constraints mentioned above, the editor checks some basic constraints, such as circularity. When modifying a large quality model, circular references are sometimes introduced inadvertently. The immediate reporting of such errors helps in keeping a model valid.

A constraint of the meta-model, not used by a wizard, is that of inheriting abstract divisions alongside restrictions. In the quality-property tree in the editor, the inherited quality-properties must be manually created. If such a quality-property is missing, the editor shows a warning.

Another important constraint, not emerging from the meta-model, but from the representation of the property hierarchy as a tree (see Figure 7.4) is the following: A quality-property must not have two children being in a restriction relation. This constraint is also checked by the editor and reported.

7.2 Quality Assessment Tool

Like the quality model editor, the tool support for conducting quality assessments is also based on the Quamoco tooling, which itself is built on the quality analysis framework ConQAT[1].

Figure 7.5 illustrates the tool-chain used for quality assessments. The quality model is created with the *quality model editor*. It serves as input to the *quality assessment tool*. Depending on the measures and tools specified in the quality model, ConQAT executes external tools such as *Findbugs* and *PMD*, or uses its built-in analysis components, such as the *Clone Detective*. The

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

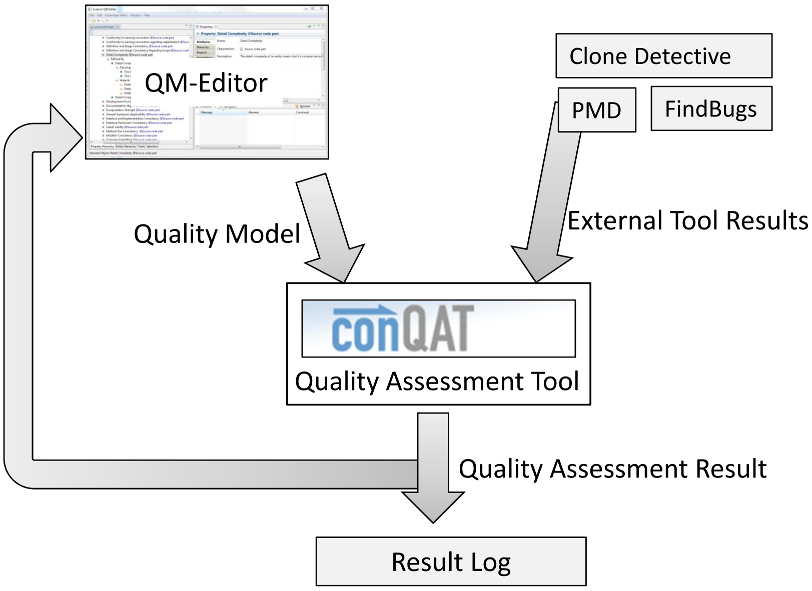
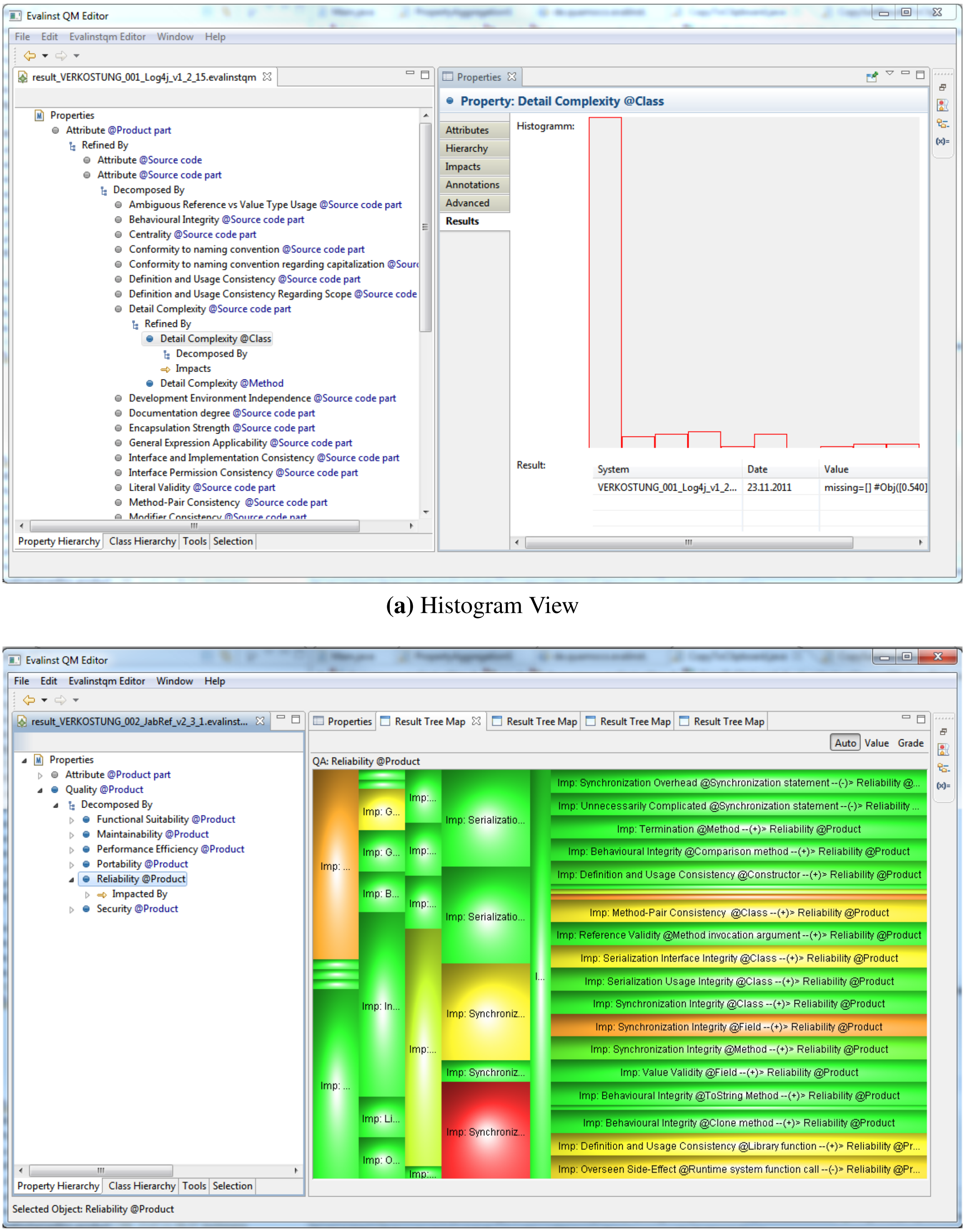


Figure 7.5: Tool-Chain for Quality Assessment

obtained measurement data is then aggregated according to the specifications of the quality model. The results are saved in a file that can be loaded in the editor for visualization. At the same time, the quality assessment tool itself produces an output with logs and visualizations of the quality assessment results.

The quality assessment tool provides the following features:

- *Complete implementation of the quality assessment approach of Chapter 5.*
  One technical challenge was to realize the concept for handling incomplete measurement data. In order to represent the potentially infinite set of possible values as a finite data structure, we chose a list of intervals. For the realization of the approach for handling findings-based measurement data of Section 5.4.3, the data structure of a *map* was used.
- *Flexible integration of analysis tools.*
  In the quality model editor, the measures and tools are modeled as references to ConQAT block names. This way, new tools can be integrated without changing the source code.
- *Visualization of Assessment Results in the Editor.*
  The result files produced by the quality assessment tool can be loaded to the quality model editor. The quality model editor provides several visualization possibilities: Figure 7.6a shows a histogram of the degree of satisfaction for all objects of a certain quality-property. Figure 7.6b shows a tree-map view of the property hierarchy. In the tree-map for a given quality-property, the rectangles denote sub-properties, whereby the area of the rectangles reflect their weight for aggregation and the color indicates the assessment value. The tree-map allows navigating the property hierarchy by double-clicking on the rectangles for sub-properties.



Properties
- Attribute @Product part
  - Refined By
    - Attribute @Source code
    - Attribute @Source code part
      - Decomposed By
        - Ambiguous Reference vs Value Type Usage @Source code part
        - Behavioural Integrity @Source code part
        - Centrality @Source code part
        - Conformity to naming convention @Source code part
        - Conformity to naming convention regarding capitalization @Sourc
        - Definition and Usage Consistency @Source code part
        - Definition and Usage Consistency Regarding Scope @Source code
        - Detail Complexity @Source code part
          - Refined By
            - Detail Complexity @Class
              - Decomposed By
              - Impacts
            - Detail Complexity @Method
        - Development Environment Independence @Source code part
        - Documentation degree @Source code part
        - Encapsulation Strength @Source code part
        - General Expression Applicability @Source code part
        - Interface and Implementation Consistency @Source code part
        - Interface Permission Consistency @Source code part
        - Literal Validity @Source code part
        - Method-Pair Consistency @Source code part
        - Modifier Consistency @Source code part

Property: Detail Complexity @Class

System	Date	Value
VERKOSTUNG_001_Log4j_v1_2...	23.11.2011	missing=[] #Obj([0.540]

(a) Histogram View

Properties
- Attribute @Product part
- Quality @Product
  - Decomposed By
    - Functional Suitability @Product
    - Maintainability @Product
    - Performance Efficiency @Product
    - Portability @Product
    - Reliability @Product
      - Impacted By
    - Security @Product

Imp: Synchronization Overhead @Synchronization statement --(-)> Reliability @...
Imp: Unnecessarily Complicated @Synchronization statement --(-)> Reliability ...
Imp: Termination @Method --(+)> Reliability @Product
Imp: Behavioural Integrity @Comparison method --(+)> Reliability @Product
Imp: Definition and Usage Consistency @Constructor --(+)> Reliability @Product
Imp: Method-Pair Consistency @Class --(+)> Reliability @Product
Imp: Reference Validity @Method invocation argument --(+)> Reliability @Product
Imp: Serialization Interface Integrity @Class --(+)> Reliability @Product
Imp: Serialization Usage Integrity @Class --(+)> Reliability @Product
Imp: Synchronization Integrity @Class --(+)> Reliability @Product
Imp: Synchronization Integrity @Field --(+)> Reliability @Product
Imp: Synchronization Integrity @Method --(+)> Reliability @Product
Imp: Value Validity @Field --(+)> Reliability @Product
Imp: Behavioural Integrity @ToString Method --(+)> Reliability @Product
Imp: Behavioural Integrity @Clone method --(+)> Reliability @Product
Imp: Definition and Usage Consistency @Library function --(+)> Reliability @Pr...
Imp: Overseen Side-Effect @Runtime system function call --(-)> Reliability @Pr...

Selected Object: Reliability @Product

(b) Tree-Map View

Figure 7.6: Visualization of Assessment Result in the Quality Model Editor

8 Case Study: Java Source Code Quality

In this section, we present a case study on the quality model for Java source code. The first research question investigates the suitability of the meta-model for creating large quality models. The next five research questions will focus on evaluating the results of automatic quality assessments using the quality model. The sixth and last research question addresses a main threat-to-validity of the other research questions. Since a benchmarking approach is used to find threshold values for the evaluation specifications in the quality model, we will show that the results of the other research questions do not depend on the chosen benchmarking base.

The first section of this chapter describes the study objects. The sections following thereafter are dedicated to one research question each.

8.1 Study Objects

The primary study object is the quality model for Java source code presented in Section 6.2.

For conducting automated quality assessments, software products to be assessed are needed. We used the repository SDS [5,119], containing about 18,000 open-source Java projects. These projects have been mostly retrieved from open source databases such as *Sourceforge* by a Web-crawling approach. In essence, this repository contains mirrors of the version control repositories of the aforementioned databases. Such repositories usually contain not only the current version of a software, but also "branches" and "tags". Thus, we use a heuristic to identify the directory containing the current version. This heuristic reads as follows in pseudo-code:

```
if not exists directory 'trunk' in recursive sub-directories then
    return root-directory;
else if there is a sub-directory 'trunk' in direct sub-directories then
    return the sub-directory
else
    for each direct sub-directory do
        repeat this algorithm
end
```

This heuristic is able to recognize the following patterns of directory structures:

1. The source code is located directly in the root directory.
2. The root directory contains the usual structure with "trunk", "branches", "tags".
3. The root directory contains exactly one directory, which contains the source code.
4. The root directory contains exactly one directory, which contains the usual structure with "trunk", "branches", and "tags".
5. The root directory contains multiple Java projects in single directories (e.g., as usually found in eclipse applications), which are then recognized as single projects.

The SDS repository only contains the source code, not the binaries. For the quality assessment however, binaries compiled with the debug-option of the Java compiler are needed. We compile all projects in a batch approach, because the effort to manually configure and compile them is prohibitive. The compilation of all 18,000 projects took about 30 hours, executed in parallel on 12 personal computers. Of all available projects, about 6,000 were compiled successfully. Others could not be compiled because of missing external libraries or because code needed to be generated during the build process.

For the studies of RQ 2, RQ 3, and RQ 4 we use all systems of the SDS repository larger than 5.000 LoC as a benchmarking base. We exclude smaller systems, because many open source repositories contain software projects initiated by single persons without finishing them; these projects then remain in the repository without ever being used [11, 127]. In RQ 5, we conduct the studies of RQ 2 to 4 with different benchmarking bases, randomly selected from the products with more than 5.000 LoC. The distribution of sizes of the used systems shows that half of the systems are smaller than 11 kLoC and 90% of the systems are smaller than 50 kLoC. Only 54 systems are larger than 100 kLoC, with the largest system having 477 kLoC.

8.2 RQ 1: Suitability for Creating Realistic Quality Models

Is the proposed quality meta-model suitable for creating realistic quality models?

We assess whether it is possible to create realistic quality models conforming to our meta-model. A main problem of creating a quality model for real-world purposes, in contrast to an illustrative example, is manage its size during creation and application. We assess whether it is possible to build a realistic quality model with the tool-support built on the meta-model.

8.2.1 Design and Procedure

First, we evaluate the tasks conducted by the quality modeler during the transformation and manual adaptation of the Quamoco Base Model to the meta-model of this thesis (see Section 6.2 for a thorough description of this process). Second, we provide some descriptive statistics used for discussing the structuredness of the resulting quality model.

1. *Elaborate Tool-Support:* The quality model editor already described in Chapter 7 is evaluated regarding its support of a set of frequently conducted scenarios. We describe the scenarios in detail and show how the quality model editor supports the user operations and/or avoids introducing modeling mistakes by the user.
2. *Building Large Quality Models:* The developed quality model is analyzed for its structuredness by descriptive statistics. We calculate the minimum/average/maximum of the following metrics: (1) number of sub-properties for component properties, (2) depth of the property hierarchy of component properties, (3) number of impacts originating from component properties, and (4) number of impacts targeting quality attributes.

3. *Completeness of the Java Quality Model:* For an estimation of how well the quality attributes are covered by component properties and measure, we estimate their completeness by counting the measures per quality attribute.

8.2.2 Study Results

The results of RQ 1 are structured according to the four criteria described in the study design.

Wizards Supporting Creation and Modification

In order to support the user in creating quality models, the quality model editor makes use of the various relations and constraints in order to support the user. In the following we describe typical scenarios and explain how the editor supports them.

Changing Property Names
The meta-model defines that a quality-property restricting another quality-property must have the same name. Thus, if the name of one quality-property is changed, then the editor changes the names of all restricting quality-properties accordingly.

Changing Classes
The meta-model defines that a quality-property dividing another quality-property must have the same class. Thus, if the class of one quality-property is changed, the editor changes the classes of all dividing quality-properties accordingly.

Selecting a Class
The meta-model defines that if a quality-property p_2 restricts a quality-property p_1, the class of p_2 must be a subclass of the class of p_1. This constraint is used by the editor to facilitate the selection of a class for a quality-property. If the quality-property restricts another quality-property, then the wizard for the selection of a class shows only subclasses of the parent-property. In many cases this wizard significantly shortens the list of classes the user has to choose from.
For instance, in the Java model the user wants to introduce a restricting quality-property for [*Definition and Usage Consistency*|*Library Statement*]. For the newly created quality-property, the user then has to choose a subclass of *Library statement*. This means choosing between 21 instead of 86 classes.

Selecting a Suitable Weighting
As described in Section 5.2.5, for rule-based static code analysis measures, we usually choose a weighting by ranges. In this case, the modeler has the task to define a suitable range-class for the given quality-property. The approach of ranges restricts possible range-classes to the parent classes of the quality-properties' class. The wizard in the editor uses this information in order to show only possible range-classes.
In the Java model, for instance, this means that instead of showing all 86 classes the editor shows typically only two or three classes.

Creating Classes
When creating a new class, the user first selects a superclass of the newly-created class. According to the inheritance of the part-of relation alongside the is-a relation, the editor then automatically sets

the newly-created class to have the same parent as its superclass. In the Java model, the inherited parent was not changed in 76.9% of the classes. Hence, in three quarters of cases the user is saved the effort of selecting the parent.

Selecting a Parent
In addition to the inheritance of the parent of classes alongside the is-a relation, the meta-model also defines that the inherited parent must only be replaced with one of its children. This constraint is used so that the wizard for changing the parent of a class only shows the allowed classes and not all classes. Like in the scenario *Selecting a Class* the number of classes to choose from is sharply reduced this way.

The Java model in Numbers

Number of Model Elements
In the following table we present the number of model elements to give an impression of the size of the model.

Model Element	**Count**
Classes	86
Quality Attributes	38
Quality Attributes with Ingoing Impacts	9
Component Properties	566
(Leaf) Component Properties with Measures	378
Component Properties with Outgoing Impacts	144
Component Properties for Structuring only	44
Impacts	370
Measures	378
Boolean Measures	369
Floating-point Measures	9

Size Metrics
For judging the structuredness of the quality model, we calculate several ratios shown in the following:

Model element	**Min.**	**Avg.**	**Max.**
Outgoing Impacts per Component Property	1	2.56	6
Ingoing Impacts per Quality Attribute	10	37.0	93
Number of Sub-Properties (Except Leaves) per Component Property	1	3.03	37
Depth of the Leaf Component Properties in the Hierarchy	2	5.01	6

Completeness of the Java Quality Model

In Table 8.1, we present the number of measures per quality attribute.

Quality Attribute	**Number of unique[a] measures**
Usability	0
Operability	0
Learnability	0
Functional Suitability	246
Functional Correctness	217
Functional Appropriateness	0
Functional Completeness	0
Time Behavior	99
Reliability	162
Security	28
Confidentiality	0
Integrity	0
Safety	0
Economic damage risk	0
Health and safety risk	0
Environmental Harm Risk	0
Maintainability	290
Analyzability	262
Modifiability	88
Verifiability	62
Releasability	0
Configurability	0
Installability	0
Reusability	45
Executability	102
Resource Utilization	102
Co-existence	0
Supportability	0
Assessability	0

[a] It is possible that one measure influences a quality attribute twice through two different impacts. For instance, the measure *number of methods per class* measures the quality-property [*Detail complexity*|*Java Class*], which has an impact on *Analyzability* and *Modifiability*. The measure is only counted once for *Maintainability*, which is divided into these two quality attributes.

Table 8.1: Completeness of the Java Model

8.2.3 Discussion

Elaborate Tool-Support
By the description of the scenarios we have seen the following advantages: If a model element is renamed, all dependent model elements are renamed too. When selecting a model element in a wizard, the tooling only shows allowed model elements. This way, the number of elements to select from is reduced by up to 90%. This strongly reduces the effort needed for selecting the right element by the user. When creating new classes, some of their attributes are set automatically using an inheritance constraint. We have shown that the inherited default value is changed in less than a quarter of cases. In this way, too, the effort for the user is reduced strongly.

Building Realistically Large Quality Models
In the quality model there are 144 component properties with outgoing impacts. These impacts are targeting only 10 quality attributes. Thus, a quality attribute is targeted by an average of 37 impacts (maximum: 93). This high number of ingoing impacts per quality attribute is seen as problematic, because the modeler may get lost in it. Yet, a component property with outgoing impacts has an average of 2.56 (maximum: 6) of them. Thus, from the viewpoint of the component properties the impacts are manageable. Furthermore, an average of 2.56 and a maximum of 6 shows that the impacts are evenly distributed.
The structuredness of the hierarchy of component properties is discussed by looking at the number of sub-properties. Each component property (leafs excluded) has an average of 3.03 sub-properties. The depth of the hierarchy averages at 5, with a maximum of 6. Thus, we conclude that the component property hierarchy is evenly distributed and balanced.
Another observation is that the Boolean measures clearly outnumber other measures. This can be explained by the fact that rule-based static code analysis tools check a large number of rules as a matter of principle.
Summing up, we conclude that the built quality model is well-structured and well-manageable.

Completeness of the Java Quality Model
The analysis of completeness shows us that the coverage of the quality attributes by measures is very diverse. Only six top-level quality attributes (functional suitability, reliability, security, maintainability, reusability, and executability) have a high coverage. Some quality attributes, such as usability, are not associated with any measure. We interpret this uneven distribution to be a consequence of using exclusively source code measures. It is understandable that maintainability and reliability mostly rely on source code properties, while usability cannot be assessed by looking at the source code.

Threats to Validity

For increasing the internal validity, we used triangulation to show the suitability of our meta-model to build realistic quality models: (1) we constructively analyzed scenarios and (2) we discussed the structuredness using descriptive statistics. The external validity is limited, because we only built one quality model for source code quality of one programming language. However, for models referring to source code quality, we assume good generalizability, because the model itself is extensive and it incorporates different types of measures.

8.3 RQ 2: Differentiation of Software Products

Do products of different quality get different quality assessment results?

Using the tool support, we conduct a quality assessment of software products. Then, we analyze the assessment results regarding *diversification*. Diversification describes whether the quality assessment yields different values for products with different quality levels.

8.3.1 Study Design and Procedure

To get an intuitive impression of the distribution of the assessment result values, we first print histograms for all quality attributes. Furthermore, we calculate the following descriptive statistics: minimum, maximum, mean, standard deviation, and quantiles.

For evaluating the diversification of software products via entropy we use a similar study design as published by Kläs et al. [89]. As prime measure for the diversification, we use the entropy. It is a measure commonly used in operational research for “diversification” [148]:

Definition 8.1 Entropy. *The entropy in information science is defined as follows:*

$$E = -\sum p_i \times ln(p_i)$$

$i = 1 \dots m$

p_i *is the probability to obtain scale level* i

An entropy of $E = 0$ means there is a probability of 100% for obtaining the same assessment results for all products; thus there is no differentiation at all. The other extreme is that all scale levels are equally distributed, resulting in a high value of E. Since the maximum value of E depends on the number of scale levels m, we define a normalized entropy yielding values between 0 and 1:

Definition 8.2 Normalized Entropy. *The normalized entropy yields values in a range between* 0 *and* 1. *It is defined as follows:*

$$e = -\frac{1}{ln(m)} \times \sum p_i \times ln(p_i)$$

The quality model yields evaluation results as floating point numbers in the range from 0 to 1. To get to discrete values, we map them to a discrete scale between 0 and 19, by applying $f(x) = ceil(x * 20)$. This way, the entropy can be calculated for the data. In order to use this equation to estimate the diversification provided by our evaluations, we have to approximate the probability values p_i for each scale level. We can do this by determining the ratio between the assessment results in the sample with level i (n_i) and the total number of results in the sample (n):

$$p_i = \frac{n_i}{n} \qquad \text{for } i = 0 \dots 19.$$

In order to define a threshold above which the diversification is considered satisfactory, an assumption about the distribution of the assessment results must be made. The visual interpretation of the histograms leads to the assumption of either a normal distribution or a logarithmic normal distribution. Therefore, we formulate two hypotheses about the distribution.

Hypothesis H_{1_0}: The quality assessment results are normal distributed.
Hypothesis H_{1_A}: The quality assessment results not are normal distributed.

For testing for normality we use a Shapiro-Wilk test [136]. This test can be applied to samples of sizes $3 < n < 5000$ and has good statistical power for small sample sizes too. We choose an alpha level of $\alpha = 0.05$ for rejecting the hypothesis of normal distribution. We use the p-value of the test, i.e., the hypothesis H_{1_0} is rejected for $p < 0.05$.

Hypothesis H_{2_0}: The quality assessment results are log-normal distributed.
Hypothesis H_{2_A}: The quality assessment results are not log-normal distributed.

We conduct a Kolmogorov-Smirnov (K-S) test [107] for this hypothesis. This test has the advantage that it can be applied not only to test for normality, but it can be used to compare a sample distribution with an arbitrary reference distribution. It is known to be very robust and reliable. To conduct the K-S test for log-normal distribution we compare the sample distribution to the assumed theoretical logarithmic normal distribution $\mathcal{LN}(\mu, \sigma)$. The parameters μ and σ of the distribution best fitting to our sample data are unknown and have to be determined. We determine these values by defining an optimization problem, searching for values for these parameters so that the p value of the K-S test is maximized. To obtain start-values for the optimization problem solver we use the following relations of the parameters μ and σ to the mean and standard deviation:

$$\begin{aligned} \mathrm{E}[X] &= e^{\mu + \frac{1}{2}\sigma^2} \\ \mathrm{s.\,d.}[X] &= e^{\mu + \frac{1}{2}\sigma^2}\sqrt{e^{\sigma^2} - 1} \end{aligned} \tag{8.1}$$

Accordingly, if $\mathrm{E}[X]$ and $\mathrm{s.\,d.}[X]$ are given, μ and σ can be estimated as follows:

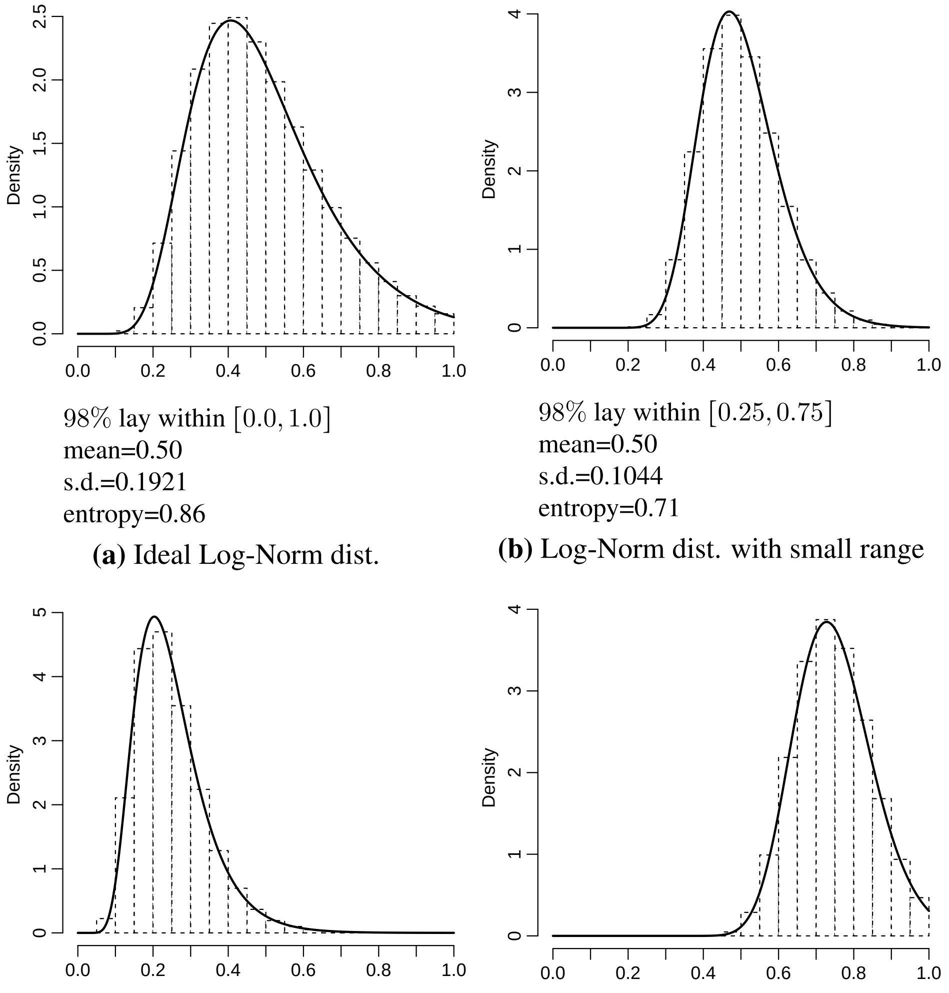
$$\begin{aligned} \mu &= \ln \mathrm{E}[X] - \frac{1}{2}\ln\left(1 + \left(\frac{\mathrm{s.\,d.}[X]}{\mathrm{E}[X]}\right)^2\right) \\ \sigma &= \sqrt{\ln\left(1 + \left(\frac{\mathrm{s.d.}[X]}{\mathrm{E}[X]}\right)^2\right)} \end{aligned} \tag{8.2}$$

We choose an alpha level of $\alpha = 0.05$ for rejecting the hypothesis H_{2_0} of logarithmic normal distribution, i.e., the hypothesis is rejected for $p < 0.05$.

Threshold for the Entropy

The assessment results are in a range of $0..1$. Thus, we assume that the ideal log-normal distribution has a mean $E[X] = 0.5$ so that it lies in the middle. The standard deviation we choose so that 98% of the log-normal function lies within the $0..1$ interval: $\int_{0.0}^{1.0} \phi(x) = 0.98$. This results in a standard deviation $s.d[X] = 0.1921$, shown in Figure 8.1a.

We allow deviations from this ideal distribution, up to different extremes. First, the mean remains at 0.50, but only half of the range $0..1$ is used, i.e., the standard deviation was chosen so that 98% lay



98% lay within $[0.0, 1.0]$
mean=0.50
s.d.=0.1921
entropy=0.86

(a) Ideal Log-Norm dist.

98% lay within $[0.25, 0.75]$
mean=0.50
s.d.=0.1044
entropy=0.71

(b) Log-Norm dist. with small range

98% lay within $[0.0, 0.5]$
mean=0.25
s.d.=0.0960
entropy=0.66

(c) Left-shifted Log-Norm dist.

98% lay within $[0.5, 1.0]$
mean=0.75
s.d.=0.1064
entropy=0.70

(d) Right-shifted Log-Norm dist.

Figure 8.1: Normal Distributions with Different Means and Ranges

within the interval $[0.25, 0.75]$. This results in a standard deviation of 0.1044, shown in Figure 8.1b. Second, we allow the mean to shift left to 0.25 and the function using the range $[0.00, 0.50]$. This log-norm function gets a standard deviation of 0.0960 and is shown in Figure 8.1c. Third, the means shifts to 0.75 and the range is $[0.50, 1.00]$, resulting in a standard deviation of 0.1064, shown in Figure 8.1d. Table 8.2 shows the entropies of these functions.

Log-Norm Distribution	Entropy
Ideal	0.86
Small Range	0.71
Left-Shifted	0.66
Right-Shifted	0.70

Table 8.2: Entropies of Log-Norm Distributions

Summing up, the distribution is regarded as sufficient if the entropy is greater or equal to 0.66.

8.3.2 Results

Figure 8.2 shows histogram plots for all quality attributes of the quality model. Table 8.3 shows the descriptive statistics for the quality attributes.

Hypothesis H_1: The quality assessment results are normal distributed.

The results of the Shapiro test for normality can be found in Table 8.4. The Shapiro test rejected the hypothesis H_{1_0} in favor of the alternate hypothesis. Therefore, we conclude that none of the quality attributes is normal distributed.

Hypothesis H_2: The quality assessment results are log-normal distributed.

The results of the K-S test for logarithmic normality can be found in Table 8.4. The hypothesis H_{1_0} of log-normality is rejected for five of twelve quality attributes.

Assessment of Differentiation by Entropy

For each quality attribute, we discretize the result values and calculate the entropy. Table 8.5 shows the results of it.

8.3.3 Discussion

The hypothesis tests showed that normal distribution of the results was rejected for all quality attributes. Logarithmic normal distribution was rejected for 5 of 12 quality attributes. Interestingly, it was not rejected for the quality attributes *Quality*, *Functional Suitability*, or *Maintainability*, which are the most complete quality attributes according to the discussion in RQ 1. Thus, we assume that

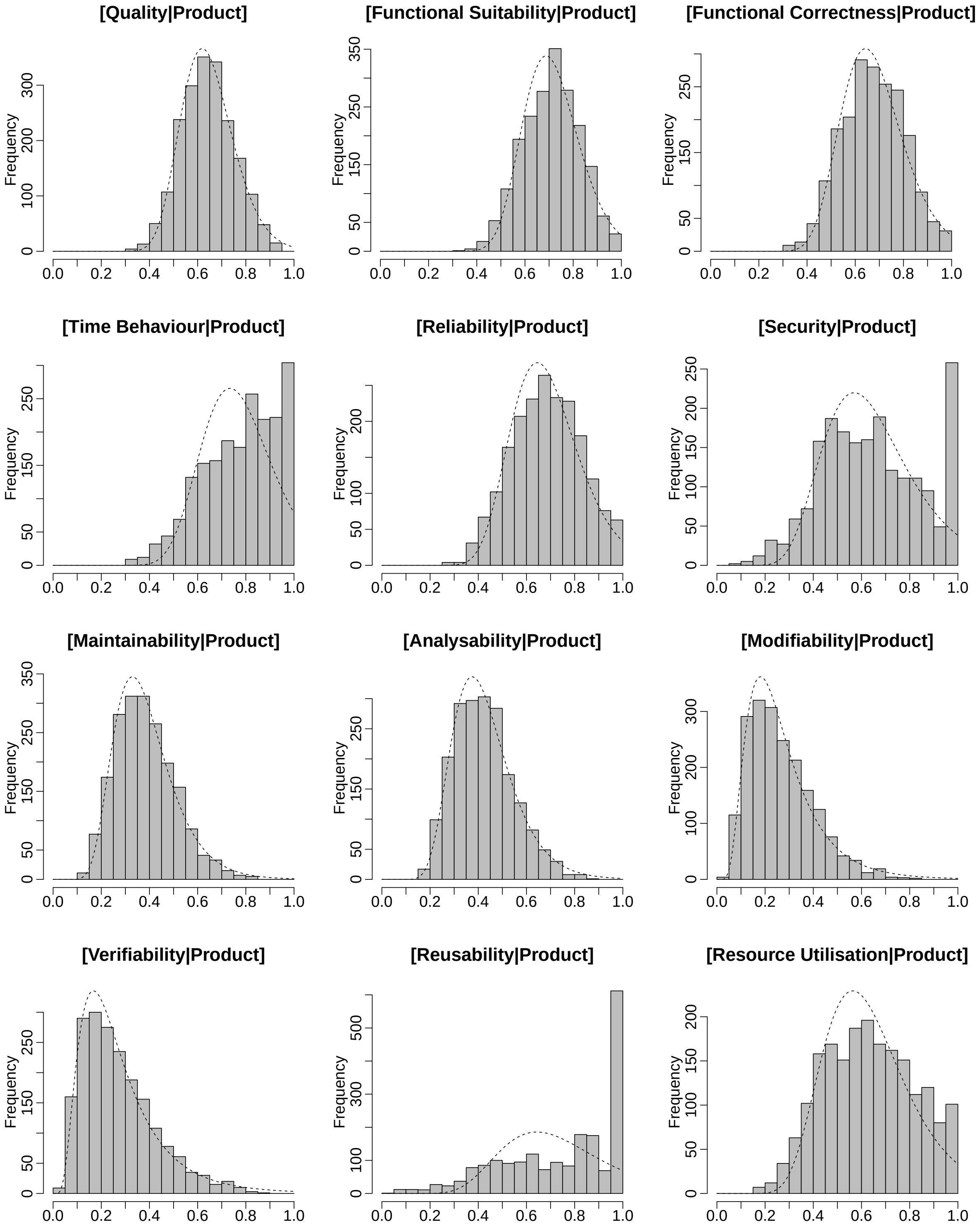


Figure 8.2: Result Histograms

Quality Attribute	Min	Q-25	Median	Q-75	Max	Mean	Std.dev.
Quality	0.32	0.56	0.64	0.72	0.93	0.64	0.1083
Functional Suitability	0.34	0.63	0.72	0.79	0.98	0.71	0.1164
Functional Correctness	0.31	0.59	0.67	0.77	0.97	0.68	0.1268
Time Behavior	0.30	0.66	0.80	0.91	1.00	0.78	0.1558
Reliability	0.26	0.58	0.68	0.79	1.00	0.68	0.1436
Security	0.07	0.48	0.63	0.81	1.00	0.65	0.2110
Maintainability	0.11	0.29	0.37	0.46	0.85	0.38	0.1233
Analyzability	0.15	0.33	0.41	0.50	0.88	0.42	0.1226
Modifiability	0.03	0.16	0.24	0.35	0.84	0.27	0.1339
Verifiability	0.04	0.16	0.24	0.35	0.85	0.27	0.1513
Reusability	0.04	0.56	0.82	1.00	1.00	0.75	0.2401
Resource Utilization	0.16	0.49	0.63	0.77	1.00	0.63	0.1870

Table 8.3: Descriptive Statistics of Quality Attributes

Quality Attribute	$H1_0$: Normality		$H2_0$: Log-Normality	
	Shapiro test p	Reject $H1_0$	K-S test p	Reject $H2_0$
Quality	0.001842	yes	0.644082	no
Functional Suitability	0.000021	yes	0.057818	no
Functional Correctness	0.000005	yes	0.062310	no
Time Behavior	0.000000	yes	0.000000	yes
Reliability	0.000002	yes	0.037704	yes
Security	0.000000	yes	0.000003	yes
Maintainability	0.000000	yes	0.688586	no
Analyzability	0.000000	yes	0.685177	no
Modifiability	0.000000	yes	0.266487	no
Verifiability	0.000000	yes	0.323833	no
Reusability	0.000000	yes	0.000000	yes
Resource Utilization	0.000000	yes	0.000291	yes

Table 8.4: Test for Normal Distribution and Log-Normal Distribution

Quality Attribute	Entropy	Sufficient Differentiation
Quality	0.7323	✓
Functional Suitability	0.7548	✓
Functional Correctness	0.7833	✓
Time Behavior	0.7970	✓
Reliability	0.8168	✓
Security	0.8907	✓
Maintainability	0.7642	✓
Analyzability	0.7630	✓
Modifiability	0.7697	✓
Verifiability	0.8018	✓
Reusability	0.8220	✓
Resource Utilization	0.8923	✓

Table 8.5: Entropy of Quality Attributes

quality attributes are log-normal distributed in general. Based on this assumption we also determined the threshold values for the test of differentiation using entropy.

The test for differentiation using the entropy showed that the entropy was higher than the defined threshold for all quality attributes. Thus, we conclude that the differentiation of the products by the quality model is sufficient.

Threats to Validity

Regarding internal validity, we do not know the “real” quality of the tested software products. Hence, it could be possible that all software products are of the same quality and our quality model should yield the same result for each of them. However, we regard this threat as minor due to the high number of used systems (approx. 2000). We see it as unlikely that such a large number of open source systems are of the same quality.

Another internal threat is the assumption of log-normality. Although the hypothesis of log-normality was not rejected for the quality attributes with the best coverage, this is no proof for the results actually being log-normal distributed. Since the thresholds for the entropy are based on the assumption of log-normality, this is a threat to validity.

A third internal threat is the usage of the entropy as a measure for differentiation. This measure could be unsuited to testing the given hypothesis. Moreover, we had to discretize the result values of the quality model, which could lead to distortion.

Regarding external validity, the generalizability of the results is limited, because we built a quality model for source code for one specific programming language. Whether these results are valid for other quality models is unclear. However, for quality models using static code analysis, we assume good generalizability, because of the large extent of the quality model.

Another threat to generalizability is the fact that only open source products have been analyzed. It is unknown if the results also apply for software products originating from other domains.

Third, the thresholds of the evaluation functions within the quality model have been determined by a benchmarking approach. The used benchmarking base may have an influence on the results of the quality assessment. To counter this threat we showed in RQ 6 that the results hold independently of the chosen benchmarking base.

8.4 RQ 3: Meaningfulness of Quality Assessment Results

Are the quality assessment results based on the quality model concordant with expert opinion?

Using the tool support, we conduct a quality assessment of software products. Then, we analyze the assessment results regarding *validity*. Validity means that the assessment results are in concordance with the results obtained by another independent assessment approach. As an independent assessment approach we use expert opinion.

8.4.1 Design and Procedure

In order to answer RQ 3, we compare the quality assessment results of the quality model, with the expert assessment of the *Linzer Softwareverkostung* [51]. There, five open source systems in the language *Java* have been assessed by 15 experts. The experts have been working in five groups. Each group inspected each software product for about one hour. The result of the inspection was the ranking of the five software products according to their quality. The goal of the inspection was to look especially at the "internal", i.e., source code quality of the products. After the inspections, the different groups discussed their results and agreed on a common ranking of the five products. Table 8.6 shows the ranking of the five products.

Next, we conduct the automated quality assessment using our quality model for the five systems. We rank the five systems according to their results for $[\mathit{Quality}|\mathit{Product}]$ and $[\mathit{Maintainability}|\mathit{Product}]$ because the focus of the expert assessment was code quality with a strong focus on maintainability. Then, we compare the rankings of the *Linzer Softwareverkostung* with the results provided by our quality model. For that, we use Spearman's rank correlation coefficient (ρ) [141]. This coefficient has a range of $[-1;1]$, whereby -1 indicates strong negative correlation and 1 a strong positive correlation. A correlation thus means there is a high consistency between the two rankings.

For testing the correlation we do a hypothesis test, with an alpha level of 0.05. We state the following null-hypothesis $H3_0$ in order to verify the alternative hypothesis $H3_A$:

Hypothesis $H3_0$: There is no correlation between the rankings provided by the experts and the quality model.

Product	Version	LoC[1]	Rank
Checkstyle	4.4	46,240	1
Log4j	1.2.15	30,676	2
RSSOwl	1.2.4	82,258	3
TV-Browser	2.2.5	125,151	4
JabRef	2.3.1	96,749	5

Table 8.6: Software Verkostung

[1] Generated code was excluded.

Product	LSV	$[\mathit{Quality}\mid\mathit{Product}]$		$[\mathit{Maintainability}\mid\mathit{Product}]$	
	Rank	Value	Rank	Value	Rank
Checkstyle 4.4	1	0.806	1	0.612	1
Log4j 1.2.15	2	0.617	2	0.414	2
RSSOwl 1.2.4	3	0.610	3	0.310	3
TV-Browser 2.2.5	4	0.505	4	0.239	4
JabRef 2.3.1	5	0.387	5	0.216	5

Table 8.7: Rankings of Software Products

Hypothesis H_{3_A}: There is a correlation between the rankings provided by the experts and the quality model.

The Spearman's rank correlation coefficient tests for the hypothesis H_{3_0}, i.e., we reject this hypothesis if $p < 0.05$.

8.4.2 Results

First, we show the quality assessment results of the five products in Table 8.7. We can see that in both cases the rankings are the same. This is reflected by Spearmans rank correlation coefficient, which is $\rho = 1$ in both cases. Moreover, with a p-value of $p = 0.01667$ the hypothesis H_{3_0} is rejected and thus the correlation is statistically significant.

8.4.3 Discussion

The result shows a perfect correlation between the quality assessment using the quality model and the expert opinion. The correlation is statistically significant for both the quality attributes *Quality* and *Maintainability*. Thus, we conclude that the quality model is able to generate quality assessments that are in line with an expert opinion.

Threats to Validity

Regarding construct validity we have to note that we tested the ranking of products provided by experts and the quality model. The ranking excludes information on the distance of quality between two systems, which could impair the accuracy of the comparison.

Regarding internal validity, we cannot guarantee that the criterion chosen for the validation, namely the expert-based quality rating, adequately represents the quality of the products.

Regarding generalizability, the same threats mainly exist as in RQ 2. First, we used a quality model for source code quality for one programming language, although an extensive one with different types of measures. Second, the result may depend on the used benchmarking base for determining the threshold values of the evaluation functions. To mitigate this threat we showed the validity of the results for different benchmarking bases in RQ 6.

A third threat to generalizability is that we only used five open-source systems for assessing the meaningfulness of the quality assessment. We regard the small number of systems as a minor threat, because the correlation showed to be statistically significant. However, the fact that we used only open-source systems may decrease the generalizability to other types of software.

8.5 RQ 4: Sensing of Quality Improvements

Are quality improvements between different versions of a software product detectable by the quality assessment based on the quality model?

An important use case for automatic quality assessments is the continuous control of quality. Thus, a quality assessment approach should be able to detect quality improvements between different versions of the same software product. We will analyze whether rather small changes in quality are detectable.

8.5.1 Design and Procedure

For this research question, we compare the quality assessment results of different versions of the same software. As a precondition for the study, we have to know for quality differences of the compared versions from an independent source. We choose the software *ConQAT*[1], because it is open source and we are able to identify points in time when quality improvements did occur.

The development of ConQAT uses the review process LEvD [29]. Using the LEvD process each source code file gets one of three states *Red*, *Yellow*, and *Green*. *Red* means that a file is currently under work by a developer. When the developer finishes work, the file gets ready-for-review and it gets the state *Yellow*. After another developer reviews the file, the developer either requires reworks and sets the file to *Red* again, or accepts the file and it becomes *Green*.

The developers of *ConQAT* reported that before releasing a version of the software, they do a "sprint" to get as many files as possible to the state *Green*. With the help of the diagram of file states shown in Figure 8.3, they were able to reconstruct the timeframes when such sprints where done. Together with the developers, three sprints were identified:

1. from 2009-07-17 to 2009-08-08
2. from 2010-01-24 to 2010-02-17
3. from 2010-10-09 to 2010-12-04

The developers generally expressed the opinion that the quality of the product increased through these sprints. By focusing on the source code style and comments, the maintainability in particular should have increased.

For studying the visibility of improvements we compare the quality assessment results for each sprint. The results of the quality model are in concordance with the experts' assessment if after

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

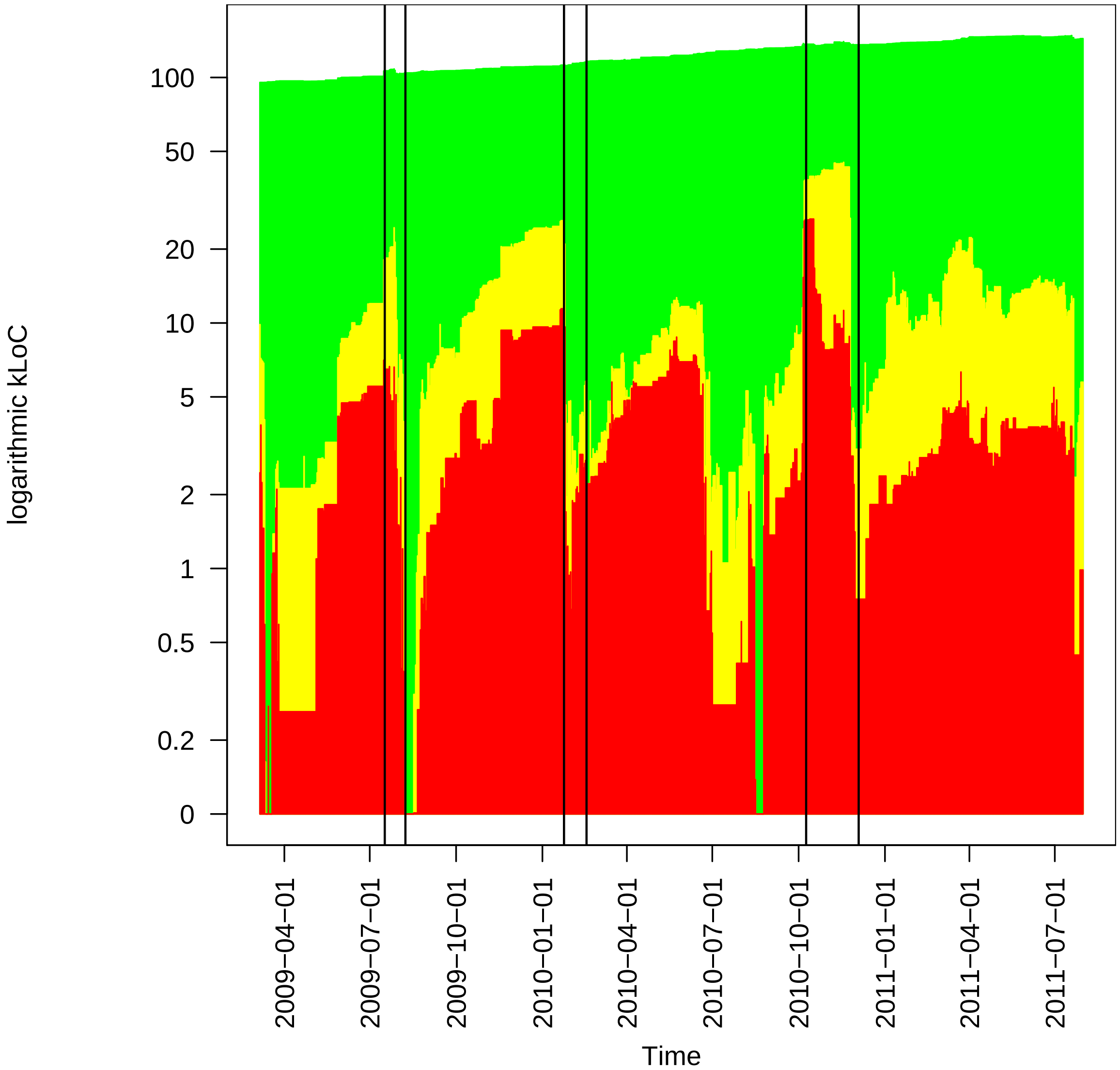


Figure 8.3: LEvD State Trend of ConQAT

each sprint an increase in quality is detected by the quality model. We do this comparison for $[Quality|Product]$, $[Maintainability|Product]$ and for its sub-properties.

8.5.2 Results

The quality assessment results of *ConQAT* are shown in Table 8.8. For $[Quality|Product]$ the values increased for two of three data points. For $[Maintainability|Product]$ the trend was only as expected for one data point.

8.5.3 Discussion

The results regarding the visibility of improvements are ambiguous. Due to the reviews and improvements conducted on the product, an improvement of the quality was expected. However, an improvement was detected by the quality model in two thirds of the cases for the quality attribute *Quality* and only in one third of the cases for many other quality attributes. Thus, we cannot conclude that the quality model is able to detect such improvements in a software product.

Threats to Validity

A major threat to the validity of this study is uncertainty in the expert opinion regarding the quality improvement. Though it is beyond question that reviews took place between the defined points in time, it is unclear to what extent reworking has been done. Moreover, it is unclear if the rework actually improved the quality. Furthermore, the experts explained that during these time periods new code was implemented, which could distort the quality assessment. Second, only a relatively small amount of the overall product has been changed at all during the review and rework periods. Thus, the extent to which the overall quality was improved is lowered again. These two factors could explain why the quality model was not able to detect an improvement altogether.

A threat to external validity is that only three improvement periods have been analyzed. Other threats to external validity are the same as in RQ 2 and RQ 3. First, the quality model is limited to source code of one programming language. Second, the used benchmarking base may have an impact on the results (see RQ 6 for an in-depth analysis of the impacts of benchmarking bases).

System	Quality	Functional Suit.	Functional Corr.	Time Beh.	Reliability	Security
2009-07-17	0.753	0.791	0.732	0.908	0.738	0.712
2009-08-08	0.783	0.815	0.756	0.933	0.765	0.705
	✓	✓	✓	✓	✓	○
2010-01-24	0.778	0.810	0.749	0.931	0.756	0.714
2010-02-17	0.756	0.800	0.734	0.930	0.741	0.650
	○	○	○	○	○	○
2010-10-09	0.758	0.777	0.684	0.962	0.739	0.708
2010-12-04	0.760	0.796	0.711	0.964	0.721	0.712
	✓	✓	✓	✓	○	✓

System	Maintainability	Analyzability	Modifiability	Verifiability	Reusability	Resource Utilization
2009-07-17	0.724	0.754	0.600	0.725	0.922	0.718
2009-08-08	0.751	0.772	0.610	0.782	0.951	0.799
	✓	✓	✓	✓	✓	✓
2010-01-24	0.738	0.758	0.604	0.766	0.931	0.789
2010-02-17	0.734	0.768	0.579	0.774	0.917	0.791
	○	✓	○	✓	○	✓
2010-10-09	0.734	0.738	0.604	0.782	0.920	0.810
2010-12-04	0.720	0.759	0.587	0.770	0.828	0.827
	○	✓	○	○	○	✓

Table 8.8: Quality Assessment Result of different Versions of ConQAT

8.6 RQ 5: Traceability of Results

Are quality improvements between different versions of a software product traceable by the quality assessment based on the quality model?

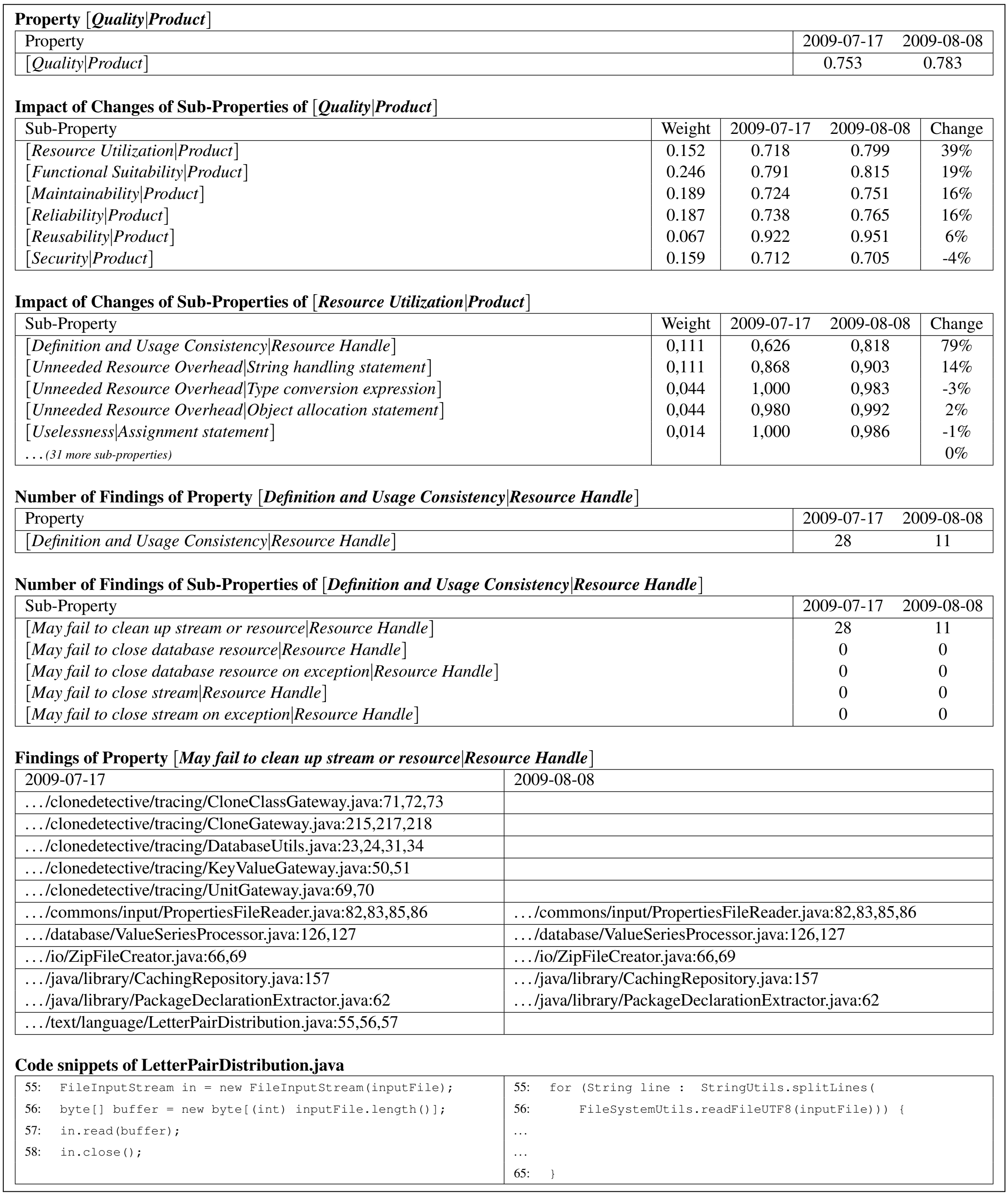
It is important that the overall quality statement generated by the quality model is traceable to individual defects in the source code. In a continuous application of the quality assessment, developers must be able to drill-down changes in the overall quality assessment results to single measures and positions in source code.

8.6.1 Design and Procedure

We use the study objects of RQ 4 for this research question. There, different versions with quality improvements/degradations have been identified. Here, we will show how a developer can trace the high-level change step-by-step to the location in the source code, which is responsible for the change.

8.6.2 Results

We do the drill-down for the first sprint (versions 2009-07-17 and 2009-08-08) in order to show where the change in the quality assessment result had its origin. Figure 8.4 shows the single steps of the drill-down, which are explained subsequently. First, we start with the quality-property $[$*Quality*$|$*Product*$]$ which increased from 0.753 to 0.783. Then, we take a look at the sub-properties of it and at the relative impacts of their changes. The relative changes are given as percentages in the table. We can see that the change of $[$*Resource Utilization*$|$*Product*$]$ had the biggest impact. Next, we take a closer look at this quality-property by analyzing its sub-properties. This quality-property has only two sub-properties, of which $[$*Resource Utilization*$|$*Product*$]$ had the bigger impact. This quality-property is influenced by 36 sub-properties via impacts. Only three of the sub-properties changed at all, with $[$*Definition and Usage Consistency*$|$*Resource Handle*$]$ having the biggest impact. This quality-property is a findings-based component property and reports objects with a value of 0, while all objects with a value of 1 are missing. Thus, we now print the number of findings for this quality-property. It decreased from 28 to 11, causing the improvement in resource utilization. $[$*Definition and Usage Consistency*$|$*Resource Handle*$]$ is divided into five other quality-properties, whose number of findings are shown in the table. Of these sub-properties only one, namely $[$*May fail to clean up stream or resource*$|$*Resource Handle*$]$ has findings at all. This quality-property is directly measured by the tool *FindBugs*. Thus, we can take a look at the findings giving locations in the source code. For one finding that appeared in the first version, but not in the second, we take a close look at the source code. In the code snippet of 2009-07-17, we can see that a *FileInputStream* is created in line 55 that is not closed if the `length()` or `read()` operation in lines 56/57 throw an exception. In the version 2009-08-08, this quality deficit was resolved by using the library function `FileSystemUtils.readFileUTF8()`.



Property [*Quality*|*Product*]

Property	2009-07-17	2009-08-08
[*Quality*|*Product*]	0.753	0.783

Impact of Changes of Sub-Properties of [*Quality*|*Product*]

Sub-Property	Weight	2009-07-17	2009-08-08	Change
[*Resource Utilization*|*Product*]	0.152	0.718	0.799	39%
[*Functional Suitability*|*Product*]	0.246	0.791	0.815	19%
[*Maintainability*|*Product*]	0.189	0.724	0.751	16%
[*Reliability*|*Product*]	0.187	0.738	0.765	16%
[*Reusability*|*Product*]	0.067	0.922	0.951	6%
[*Security*|*Product*]	0.159	0.712	0.705	-4%

Impact of Changes of Sub-Properties of [*Resource Utilization*|*Product*]

Sub-Property	Weight	2009-07-17	2009-08-08	Change
[*Definition and Usage Consistency*|*Resource Handle*]	0,111	0,626	0,818	79%
[*Unneeded Resource Overhead*|*String handling statement*]	0,111	0,868	0,903	14%
[*Unneeded Resource Overhead*|*Type conversion expression*]	0,044	1,000	0,983	-3%
[*Unneeded Resource Overhead*|*Object allocation statement*]	0,044	0,980	0,992	2%
[*Uselessness*|*Assignment statement*]	0,014	1,000	0,986	-1%
... *(31 more sub-properties)*				0%

Number of Findings of Property [*Definition and Usage Consistency*|*Resource Handle*]

Property	2009-07-17	2009-08-08
[*Definition and Usage Consistency*|*Resource Handle*]	28	11

Number of Findings of Sub-Properties of [*Definition and Usage Consistency*|*Resource Handle*]

Sub-Property	2009-07-17	2009-08-08
[*May fail to clean up stream or resource*|*Resource Handle*]	28	11
[*May fail to close database resource*|*Resource Handle*]	0	0
[*May fail to close database resource on exception*|*Resource Handle*]	0	0
[*May fail to close stream*|*Resource Handle*]	0	0
[*May fail to close stream on exception*|*Resource Handle*]	0	0

Findings of Property [*May fail to clean up stream or resource*|*Resource Handle*]

2009-07-17	2009-08-08
.../clonedetective/tracing/CloneClassGateway.java:71,72,73	
.../clonedetective/tracing/CloneGateway.java:215,217,218	
.../clonedetective/tracing/DatabaseUtils.java:23,24,31,34	
.../clonedetective/tracing/KeyValueGateway.java:50,51	
.../clonedetective/tracing/UnitGateway.java:69,70	
.../commons/input/PropertiesFileReader.java:82,83,85,86	.../commons/input/PropertiesFileReader.java:82,83,85,86
.../database/ValueSeriesProcessor.java:126,127	.../database/ValueSeriesProcessor.java:126,127
.../io/ZipFileCreator.java:66,69	.../io/ZipFileCreator.java:66,69
.../java/library/CachingRepository.java:157 .../java/library/PackageDeclarationExtractor.java:62	.../java/library/CachingRepository.java:157 .../java/library/PackageDeclarationExtractor.java:62
.../text/language/LetterPairDistribution.java:55,56,57	

Code snippets of LetterPairDistribution.java

55: `FileInputStream in = new FileInputStream(inputFile);` 56: `byte[] buffer = new byte[(int) inputFile.length()];` 57: `in.read(buffer);` 58: `in.close();`	55: `for (String line : StringUtils.splitLines(` 56: `FileSystemUtils.readFileUTF8(inputFile))) {` ... ... 65: `}`

Figure 8.4: Drill-Down for the versions 2009-07-17 and 2009-08-08

8.6.3 Discussion

The drill-down showed that the structure of the model and the tooling enable an effective traceability of quality assessment results. It is easily traceable where the change of the overall result originated from. In the example showed above, the static code analysis rule responsible for the biggest change in the result could be easily detected and the relevant source code snippets could be found.

Threats to Validity

A threat to validity for this research question is that it was not conducted in an industrial setting with developers. In practice, several factors, e.g., the integration of the approach into the development environment, play a substantial role in the usefulness of an approach. Furthermore, only one data point was analyzed.

8.7 RQ 6: Impact of Different Benchmarking Bases

How do different benchmarking bases influence the results of RQ 2, RQ 3, and RQ 4?

Our quality assessment approaches relies on defining thresholds for measurement values by an approach similar to benchmarking. Measurement values are calculated for all software products of a given benchmarking base. Statistical values, such as mean, median, and quantiles, of those data are used as threshold values. We analyze how the selection of different benchmarking bases influences the results of RQ 2, RQ 3, and RQ 4.

8.7.1 Design and Procedure

To analyze the impact of different benchmarking bases on the results of RQ 2, RQ 3, and RQ 4, we repeat those studies with the following benchmarking bases: First, we randomly select 500 products from all available products with more than 5.000 LoC. Second, we use large products (LoC > 100.000), and third, we calibrate with small products (10.000 < LoC < 20.000).

The procedure is as follows: The quality model is calibrated with these benchmarking bases. Then, the studies of RQ 2, RQ 3, and RQ 4 are repeated. Finally, we compare the results and discuss the influence of the benchmarking bases.

8.7.2 Results

The benchmarking bases used were:

A	Model calibrated with all products with $LoC > 5.000$	2041 products
B	Model calibrated with randomly selected products	500 products
C	Model calibrated with large products ($LoC > 100.000$)	65 products
D	Model calibrated with small products ($10.000 < LoC < 20.000$)	607 products

Regarding RQ 2, we show the results of the test for log-normality in Table 8.9. We can see that the results are the same for all benchmarking bases, except for the value of *Functional correctness* in benchmarking base C. In Table 8.10, we see the values for the entropy for all quality attributes and benchmarking bases. Here we can see that the result is the same for all benchmarking bases; the entropy is above the defined threshold for all quality attributes.

Regarding RQ 3, the rankings of the five products for each benchmarking base are shown in Table 8.11. We see that in two cases ($[Quality|Product]$ for benchmarking bases B and C) two products are interchanged in the ranking. For $[Maintainability|Product]$ all rankings by the quality model correspond to the expert based rankings. If the rankings correspond, Spearman's rank correlation coefficient is $\rho = 1$, with a $p = 0.01667$, thus showing statistical significance. If two ranks are interchanged, then $\rho = 0.9$, with a $p = 0.08333$, thus not being statistically significant.

Regarding RQ 4, the values for the quality attributes with the highest coverage are shown in Table 8.12. We see that the results are the same for all benchmarking bases.

8.7.3 Discussion

We can see that the results of RQ 2, RQ 3 and RQ 4 are valid for all tested benchmarking bases:

- *RQ 2:* The entropy was above the defined thresholds for all quality attributes and all benchmarking bases. Hence, the result is exactly the same for all benchmarking bases.
- *RQ 3:* Although the absolute values yielded by the quality model have changed slightly, the correlation between the expert ranking and the quality model based ranking remains high. For the quality attribute *Quality* two benchmarking bases resulted in a swap of two ranks. The correlation is still high for these ranks, but the p-value is at 0.083 and thus the result is not statistically significant. For the quality attribute *Maintainability* the rankings for all benchmarking bases correspond to the expert ranking. With a p-value of 0.01667 this correlation is statistically significant.
- *RQ 4:* The results for all benchmarking bases are exactly the same. Thus, we conclude that the different benchmarking bases have no influence on the result of RQ 4.

Additionally, we observed that the values of benchmarking base C diverge strongly from the other values. We suspect this is because benchmarking base C is both the smallest one (only 65 products) and comprises large systems, while the products under investigation are mostly small products. Thus, this benchmarking base could be less suited to the products under investigation.

Threats to Validity

Our conclusions regarding different benchmarking bases are not statistically significant due to the small number of different benchmarking bases used. To counter this threat we selected very distinct benchmarking bases (small products vs. large products). Moreover, the different benchmarking bases have a different size. The size of the benchmarking base itself may have an influence on the result.

As in RQ 2, RQ 3, and RQ 4, a threat to generalizability is again that only one quality model for source code of one programming language was used. Furthermore, we only used open-source systems as study objects. Thus, the results cannot be generalized to other quality models or programming languages. However, for this quality model, we assume the benchmarking base to have only a minor influence on the quality assessment results.

Quality Attribute	K-S test				Reject H_{2_0}			
	A	B	C	D	A	B	C	D
Quality	0.644082	0.617551	0.106442	0.719777	no	no	no	no
Functional Suitability	0.057818	0.057837	0.007675	0.100571	no	no	yes	no
Functional Correctness	0.062310	0.054119	0.007377	0.069281	no	no	yes	no
Time Behavior	0.000000	0.000000	0.000000	0.000000	yes	yes	yes	yes
Reliability	0.037704	0.030395	0.001986	0.022450	yes	yes	yes	yes
Security	0.000003	0.000003	0.000000	0.000000	yes	yes	yes	yes
Maintainability	0.688586	0.517064	0.053511	0.512512	no	no	no	no
Analyzability	0.685177	0.440128	0.163982	0.553180	no	no	no	no
Modifiability	0.266487	0.192403	0.067813	0.152365	no	no	no	no
Verifiability	0.323833	0.346812	0.074308	0.313417	no	no	no	no
Reusability	0.000000	0.000000	0.000000	0.000000	yes	yes	yes	yes
Resource Utilization	0.000291	0.001153	0.000290	0.000593	yes	yes	yes	yes

Table 8.9: RQ 2 for Different Benchmarking Bases (Test for Logarithmic Normal Distribution)

Quality Attribute	Entropy				Sufficient Differentiation			
	A	B	C	D	A	B	C	D
Quality	0.7323	0.7302	0.7996	0.7317	✓	✓	✓	✓
Functional Suitability	0.7548	0.7550	0.8375	0.7572	✓	✓	✓	✓
Functional Correctness	0.7833	0.7811	0.8536	0.7843	✓	✓	✓	✓
Time Behavior	0.7970	0.8021	0.8889	0.8062	✓	✓	✓	✓
Reliability	0.8168	0.8104	0.8987	0.8176	✓	✓	✓	✓
Security	0.8907	0.8829	0.9077	0.8846	✓	✓	✓	✓
Maintainability	0.7642	0.7588	0.8051	0.7713	✓	✓	✓	✓
Analyzability	0.7630	0.7594	0.7833	0.7613	✓	✓	✓	✓
Modifiability	0.7697	0.7595	0.8354	0.7817	✓	✓	✓	✓
Verifiability	0.8018	0.7916	0.7965	0.8103	✓	✓	✓	✓
Reusability	0.8220	0.8262	0.8609	0.8189	✓	✓	✓	✓
Resource Utilization	0.8923	0.8925	0.9336	0.8931	✓	✓	✓	✓

Table 8.10: RQ 2 for Different Benchmarking Bases (Entropy of Quality Attributes)

Product	LSV Rank	[*Quality*|*Product*]								[*Maintainability*|*Product*]							
		Value				Rank				Value				Rank			
		A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
Checkstyle 4.4	1	0.807	0.800	0.790	0.817	1	1	1	1	0.612	0.602	0.587	0,628	1	1	1	1
Log4j 1.2.15	2	0.617	0.607	0.474	0.646	2	3	3	2	0.414	0.399	0.361	0.432	2	2	2	2
RSSOwl 1.2.4	3	0.610	0.611	0.506	0.623	3	2	2	3	0.310	0.310	0.255	0.306	3	3	3	3
TV-Browser 2.2.5	4	0.505	0.498	0.369	0.515	4	4	4	4	0.239	0.226	0.167	0.249	4	4	4	4
JabRef 2.3.1	5	0.387	0.364	0.248	0.395	5	5	5	5	0.216	0.207	0.157	0.219	5	5	5	5

Table 8.11: RQ 3 for Different Benchmarking Bases (Rankings of Software Products)

System	Quality				Functional Suitability				Maintainability				Reliability			
	A	B	C	D	A	B	C	D	A	B	C	D	A	B	C	D
2009-07-17	0.761	0.759	0.637	0.773	0.791	0.718	0.651	0.746	0.724	0.701	0.705	0.744	0.738	0.776	0.653	0.772
2009-08-08	0.784	0.782	0.665	0,794	0.815	0.748	0.692	0.768	0.751	0.730	0.773	0.766	0.765	0.800	0.708	0.801
	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
2010-01-24	0,778	0,775	0,646	0,789	0,810	0,743	0,646	0,760	0,738	0,715	0,755	0,754	0,756	0,782	0,660	0,793
2010-02-17	0,756	0,753	0,590	0,764	0,800	0,719	0,618	0,733	0,734	0,713	0,726	0,749	0,741	0,768	0,626	0,774
	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
2010-10-09	0,758	0,757	0,610	0,769	0,777	0,682	0,568	0,693	0,734	0,712	0,733	0,749	0,739	0,766	0,640	0,775
2010-12-04	0,760	0,760	0,614	0,772	0,796	0,710	0,639	0,721	0,720	0,700	0,680	0,735	0,721	0,749	0,608	0,757
	✓	✓	✓	✓	✓	✓	✓	✓	○	○	○	○	○	○	○	○

Table 8.12: RQ 4 for Different Benchmarking Bases (Visibility of Improvement)

9 Conclusion

Despite the importance of software quality, there are several problems regarding its definition and assessment. There is a gap between quality models for defining quality and software measurement approaches. On one hand, there are quality models defining quality attributes, which are not precise enough to be directly measurable. On the other hand, software measurement approaches provide a large number of measures for characteristics of software products. However, the relation of these measures to the quality models is unclear. Thus, these measures are applied in an inconsistent manner and overall quality statements for software products cannot be issued. Moreover, the definition of quality attributes by quality models is often unclear and ambiguous, thus further limiting their acceptance in practice.

In this thesis, we propose a quality modeling and assessment approach bridging the gap between quality definition and assessment. Our approach enables the construction of quality models, which define quality attributes and their relation to component properties, which are directly measurable. The quality model is operationalized by providing a specification of how measurement results are aggregated to an overall quality statement.

Defining Quality

First, we introduce an explicit quality meta-model defining the structure of quality models. The meta-model formally describes the elements of which quality models consist. This way, we achieve a clear and unambiguous definition of the meaning of quality model elements.

In the literature, there are several approaches that directly operationalize quality attributes by attaching measures to them. Since those approaches proved unsatisfactory, we introduce an intermediate layer between quality attributes and measures: *component properties*. They describe characteristics of artifacts the software product consists of. For each component property we define impacts on quality attributes, describing and justifying the relation of the component property to the quality attribute. To define the component properties in a clear and unambiguous way, we use a *product model* of the software system. It describes the artifacts of the software product in the form of a data model with generalization and composition relations known from class diagrams.

The structuring mechanism for component properties still does not address the clear definition of quality attributes. In order to overcome the unclear and overlapping definitions of quality attributes, we rely on the principle of activity-based quality models. We define an activity-hierarchy describing activities conducted with the software product during its lifecycle and derive a hierarchy of quality attributes from them. Since activities can be clearly decomposed into sub-activities, the quality attributes defined based on activities are inherently unambiguous.

Assessing Quality

To close the gap between quality models and software measurement, we operationalize our quality model by integrating existing measures and their implementations by tools. The measures are attached to component properties and are used to quantify the satisfaction of the respective properties. Thereby, we overcome the problem of using measures without a clear rationale. For each measure its connection to a quality attribute via impacts and component properties clearly defines its relation to quality. For actually quantifying quality attributes and for delivering an overall quality statement for a software product, the quality model defines how the measurement data is aggregated. The aggregation follows the well-defined structure of the component properties and quality attributes. This way, the formation of the overall quality statement is explained by the quality model. Hence, the shortcoming of unsystematic aggregation is overcome.

Application in Practice

To show the applicability of our quality modeling and assessment approach, we develop a concrete operationalized quality model. Since the effort for developing an operationalized quality model covering all quality attributes is prohibitive, we focus on a topic we already have experience with: source code quality with regard to maintainability. To be specific, we built a quality model for source code in the Java programming language. The quality model contains 566 component properties of which three quarters are directly measured. The measures include classical measures, such as nesting depth, but also a large number of static code analysis rules. Limiting the quality model to source code has the effect that five out of eleven top-level quality attributes are not covered by it; while maintainability is operationalized by 291 measures, usability is not operationalized at all.

Operationalizing this quality model includes addressing challenges appearing in practice. To this end, the quality assessment approach contributes the following: First, it provides an approach for defining parameters of utility functions – i.e., threshold values for measurement values – based on the principle of benchmarking. Second, it defines how incomplete measurement data can be handled. Third, it enables the integration of existing tools by calculating the aggregation bottom-up. Fourth, it defines how rule-based static code analysis tools yielding values on a nominal scale are integrated with measures on ratio scales.

Evaluation

The quality assessment approach is evaluated using the quality model for Java source code. As study objects, we use 2041 open source Java systems. First, we show that the differentiation of the software systems by the quality assessment results is satisfactory. This means that the quality assessment yields different values for software products with different quality levels. Second, we test the meaningfulness of the quality assessment results by comparing it to an expert assessment. The ranking of five systems regarding their quality is the same for the expert assessment and the quality model-based assessment. Thus, we conclude that our approach is able to produce valid quality assessment results. Third, we show by example that the quality assessment based on the quality model enables the tracing of quality changes between different versions of a software system. This is important for enabling developers to identify the origin of a change in the quality assessment.

10 Future Work

The quality modeling and assessment approach presented in this thesis addresses several problems in the area of software quality. However, software quality is a very broad topic and a large number of challenges remain unsolved. In this chapter, we discuss possible directions for further research.

10.1 Further Evaluation

In this thesis, we used exclusively open source systems to evaluate the quality assessment approach. Additional benefit would be achieved by diversifying the study objects by using software from different industries. For increasing generalizability, different types of software, such as embedded systems or business information systems, should also be used. However, it is not only the software systems themselves that are important study objects, but also expert-based assessments of the systems for validating the quality model-based assessment result.

Besides extending the study objects, other research questions can be taken into account. For case studies conducted in industry, it would be interesting to assess the usability and acceptance of the approach by practitioners. Furthermore, cost-benefit analyses of the application of a quality model in quality assurance could be conducted.

10.2 In-Depth Analysis of the Existing Quality Model

The quality model built for Java source code and the accompanying tool support for conducting automated quality assessments gives the possibility of conducting further studies and experiments.

An interesting question is the influence of different benchmarking bases on the quality assessment results. We already took a first step in this direction in our paper [100]. We conducted a series of experiments, using different benchmarking bases. Our initial results show that the larger the benchmarking base, the less divergent are the quality assessment results obtained.

Further studies of this type could investigate the influence of other criteria than size, such as the domain for which the systems have been developed. Seen from a more universal perspective, using different benchmarking bases is just one way of modifying the quality model. A similar study could investigate the sensitiveness of the quality assessment results to modification of the quality model. The sensitiveness to modification is a relevant issue for estimating the transferability and comparability of assessment results. This, in turn, is a relevant question for providing quality certificates based on the quality assessment, since comparability is a fundamental prerequisite for certifying software products of different domains and industries.

10.3 Broader Scope of the Quality Model

The presented quality meta-model and the general quality model defining quality attributes are widely applicable for describing software quality. The assessment approach, however, is already more specific and addresses specific problems encountered when assessing source code. The quality model for Java source code, of course, is even more specialized. There are different possibilities for creating quality models with a broader scope.

Besides source code, in software development a lot of other development artifacts are created that are relevant for quality assessments. An obvious candidate is the architecture of a software system. In software engineering a lot of work has been dedicated to architecture evaluation (e.g. [4,81]). This body of knowledge could be integrated into a quality model. The predefined structure of the quality model could lead to a consistent description of the knowledge gathered by different architecture evaluation approaches. In the area of service-oriented architectures (SOA), we have already taken a first step in building a quality model for describing SOA-specific architecture characteristics [47].

A clear limitation of our quality model describing Java source code is that it does not address usability at all. To address usability, quality characteristics of user interfaces could be modeled. In fact, during the development of the quality meta-model, we experimented with modeling an excerpt of the *Web Content Accessibility Guidelines* (WCAG) 2.0 [168]. These first experiments led to promising results. The structure of the WCAG exhibits similarities to the structure of the quality model. For instance, the four chapters of WCAG directly correspond to activities of the software usage, which can be mapped to quality attributes directly. The single sections of WCAG describe characteristics of user interfaces, which can be directly mapped to component properties. Moreover, the WCAG defines so-called techniques, which correspond to the measures of the quality model. Integrating the WCAG standard into one quality model could lead to synergy effects. For instance, once a quality model covering the entire WCAG is built, the quality assessment approach can be applied to it, enabling (semi-)automated assessments of user interfaces. This would be a considerable novelty, since currently the WCAG does not define how to assess the compliance of a product to it.

Besides the two examples presented above for extending the quality model to other artifacts than source code, a large variety of other topics could be addressed. The quality of test cases for software products as well as dynamical aspects, such as performance or reliability, could be modeled by a quality model.

10.4 Application in Practice

To apply the approach of this thesis in practice integration into an actual software development process is necessary. The role of the quality model in the different phases of the development process poses several new research questions.

Requirements Engineering

One research question is the role of the quality model for requirements engineering. Since the quality-properties defined in the quality model are potential requirements, there is obviously a strong interrelation between the requirements of a project and the quality model. On one side, the quality model has to be adapted to the needs of the project. On the other hand, the quality model serves as a repository of potential requirements that can support the requirements elicitation.

The idea of integrating an activity-based quality model into a requirements engineering process was first proposed by Wagner et al. [156, 157]. In a case study, we applied this approach retrospectively to security requirements for the Apache Tomcat Webserver [163]. Later, Luckey et al. [105] applied an activity-based quality model for reusing quality requirements in an industrial case study. Finally, we adapted the quality requirements approach to an earlier version of the meta-model proposed in this thesis and applied it in an industrial case study at Siemens [103]. We re-specified requirements in retrospective with our approach, and compared the produced specification with the legacy specification. While the completeness, structuredness, traceability, and modularity of the quality model-based specification was better than that of the legacy specification, the perceived productivity of the approach did not improve.

These first results indicate that a quality model can be applied in requirements engineering beneficially. Nonetheless, further case studies with a larger variety of study objects and subjects are necessary. This way, the current results could be further confirmed, or additional needs and suggestions for improvement could arise.

Besides conducting further case studies in the field of requirements engineering, the quality model should be introduced as a first-class component of the development process. In requirements engineering, the quality model would no longer be a supplementary artifact, but the actual repository containing the approved quality requirements. This way, the quality model would be tailored or supplemented in the requirements phase and then be used in quality assurance as outlined in Section 2.2.5.

Integration into Quality Assurance Processes

As explained in Section 2.2.5 quality models should play a central role in the quality assurance process. To actually accomplish this vision, a tight integration into the quality assurance process is necessary. While the process integration is not part of this work, this thesis laid the groundwork by providing the quality assessment approach. The process integration of the quality modeling and assessment approach includes, for instance, connecting the quality assessment with a change management system. Quality deficits identified in quality assessments must be tracked and communicated to developers. Furthermore, a cost-benefit analysis is necessary for deciding which quality defects should be corrected.

Standardization and Certification

In this thesis, we have seen that a fixed quality model can be successfully used for quality assessments. The preliminary results of using predefined quality models for requirements engineering are promising as well. This naturally leads to the question of standardizing a quality model. If quality models are applicable in different contexts, then a general quality model could be standardized by a national or international standardization organization. A standardized quality model, which is fully operationalized, would go far beyond the current standards. It could be the base for certifications of software products. As mentioned before, further work on the quality assessment approach would be necessary, to ensure transferable and comparable quality assessment results. This, of course, is a necessary precondition for certifications.

Bibliography

[1] H. Al-Kilidar, K. Cox, and B. Kitchenham. The use and usefulness of the ISO/IEC 9126 quality standard. In Proc. of the *International Symposium on Empirical Software Engineering (ISESE '05)*. IEEE Computer Society, November 2005.

[2] T. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In Proc. of the *International Conference on Software Maintenance (ICSM '10)*. IEEE Computer Society, 2010.

[3] Artistic Style. http://astyle.sourceforge.net/, last accessed on 11.07.2012, 2012.

[4] M. Babar, L. Zhu, and R. Jeffery. A framework for classifying and comparing software architecture evaluation methods. In Proc. of the *Australian Software Engineering Conference (ASWEC '04)*. ACS, April 2004.

[5] S. Bajracharya, J. Ossher, and C. Lopes. Sourcerer: An internet-scale software repository. In Proc. of the *Workshop on Search-Driven Development–Users, Infrastructure, Tools and Evaluation (SUITE '09)*. IEEE Computer Society, June 2009.

[6] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17, 2002.

[7] H. F. Barron and B. E. Barrett. Decision Quality Using Ranked Attribute Weights. *Management Science*, 42(11):1515–1523, 1996.

[8] V. Basili and H. Rombach. The TAME project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):758–773, 1988.

[9] V. Basili and R. Selby. Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, SE-13(12):1278–1296, 1987.

[10] V. R. Basili and D. M. Weiss. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering*, SE-10(6):728–738, 1984.

[11] K. Beecher, A. Capiluppi, and C. Boldyreff. Identifying exogenous drivers and evolutionary stages in FLOSS projects. *Journal of Systems and Software*, 82(5):739–750, 2009.

[12] P. Berander and P. Jönsson. A goal question metric based approach for efficient measurement framework definition. In Proc. of the *International Symposium on Empirical Software Engineering and Measurement (ESEM '06)*. ACM, 2006.

[13] K. Bittner and I. Spence. *Use case modeling*. Addison-Wesley, Boston, Mass., 8. print. edition, 2006.

[14] B. W. Boehm. *Characteristics of Software Quality*. North-Holland, 1978.

[15] G. Böhme. *Fuzzy-Logik: Einführung in die algebraischen und logischen Grundlagen*. Springer-Lehrbuch. Springer Verl, Berlin u.a, 1993.

[16] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the Unified Modeling Language. In Proc. of the *European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1997.

[17] M. Broy, M. V. Cengarle, B. Rumpe, M. Crane, J. Dingel, Z. Diskin, J. Jürjens, and B. Selic. *Semantics of UML. Towards a System Model for UML. The Structural Data Model: Technical Report of Technische Universität München*. Number TUM-I0612. 2006.

[18] M. Broy, M. Feilkas, J. Grünbauer, A. Gruler, A. Harhurin, J. Hartmann, B. Penzenstadler, B. Schätz, and D. Wild. *Umfassendes Architekturmodell für das Engineering eingebetteter Software-intensiver Systeme: Technical Report of Technische Universität München*. Number TUM-I0816. 2008.

[19] M. Broy, M. Jarke, M. Nagl, and H. D. Rombach. Manifest: Strategische Bedeutung des Software Engineering in Deutschland. *Informatik Spektrum*, 29(3):210–221, 2006.

[20] Checkstyle. 5.5, http://checkstyle.sourceforge.net/, last accessed on 11.07.2012, 2012.

[21] A. Cockburn. Goals and Use Cases. *Journal of Object-Oriented Programming (JOOP)*, 10(5):35–40, 1997.

[22] M.-A. Côté, W. Suryn, and E. Georgiadou. In search for a widely applicable and accepted software quality model for software quality engineering. *Software Quality Journal*, 15(4):401–416, 2007.

[23] D. Darcy, R. Smith, C. Kemerer, S. Slaughter, and J. Tomayko. The structural complexity of software - an experimental text. *IEEE Transactions on Software Engineering*, 31(11), 2005.

[24] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[25] R. Darimont and A. van Lamsweerde. Formal refinement patterns for goal-driven requirements elaboration. *SIGSOFT Software Engineering Notes*, 21(6):179–190, 1996.

[26] F. Deissenboeck. Continuous Quality Control of Long-Lived Software Systems, PhD thesis, Technische Universität München. 2009.

[27] F. Deissenboeck, L. Heinemann, M. Herrmannsdoerfer, K. Lochmann, and S. Wagner. The Quamoco Tool Chain for Quality Modeling and Assessment. In Proc. of the *International Conference on Software Engineering (ICSE '11)*. ACM, May 2011.

[28] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens. Flexible architecture conformance assessment with ConQAT. In Proc. of the *International Conference on Software Engineering (ICSE '10)*. ACM, May 2010.

[29] F. Deissenboeck, U. Hermann, E. Juergens, and T. Seifert. LEvD: A Lean Evolution and Development Process, http://conqat.cs.tum.edu/download/levd-process.pdf, last accessed on 11.07.2012. 2007.

[30] F. Deissenboeck, B. Hummel, and E. Juergens. ConQAT - Ein Toolkit zur kontinuierlichen Qualitätsbewertung. In Proc. of the *Software Engineering Konferenz (SE '08)*. GI, February 2008.

[31] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, Benedikt Mas y Parareda, and M. Pizka. Tool Support for Continuous Quality Control. *IEEE Software*, 25(5):60–67, 2008.

[32] F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner. Software quality models: Purposes, usage scenarios and requirements. In Proc. of the *Workshop on Software Quality (WoSQ '09)*. IEEE Computer Society, May 2009.

[33] F. Deissenboeck, Stefan Wagner, M. Pizka, S. Teuchert, and J.-F. Girard. An Activity-Based Quality Model for Maintainability. In Proc. of the *International Conference on Software Maintenance (ICSM '07)*. IEEE Computer Society, October 2007.

[34] R. G. Dromey. A model for software product quality. *IEEE Transactions on Software Engineering*, 21(2):146–162, 1995.

[35] R. T. Eckenrode. Weighting Multiple Criteria. *Management Science*, 12(3):180–192, 1965.

[36] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., 1998.

[37] FindBugs. Find Bugs in Java Programs, http://findbugs.sourceforge.net/, last accessed on 11.07.2012.

[38] R. France, J.-M. Bruel, M. Larrondo-Petrie, E. Grant, and M. Saksena. Towards a rigorous object-oriented analysis and design method. In Proc. of the *International Conference on Formal Engineering Methods (ICFEM'97)*. IEEE Computer Society, November 1997.

[39] X. Franch and J. P. Carvallo. Using quality models in software package selection. *IEEE Software*, 20(1):34–41, 2003.

[40] B. Freimut, L. Briand, and F. Vollei. Determining inspection cost-effectiveness by combining project data and expert opinion. *IEEE Transactions on Software Engineering*, 31(12):1074–1092, 2005.

[41] D. Galin. *Software quality assurance: From theory to implementation*. Pearson/Addison Wesley, Harlow, 2004.

[42] W. O. Galitz. *The essential guide to user interface design: An introduction to GUI design principles and techniques*. Wiley technology publishing. Wiley, Indianapolis Ind. u.a, 3 edition, 2007.

[43] D. A. Garvin. What does product quality really mean. *Sloan Management Review*, 26(1):25–43, 1984.

[44] T. Gilb, D. Graham, and S. Finzi. *Software inspection*. Addison-Wesley, Wokingham, England, 1993.

[45] M. Glinz. On Non-Functional Requirements. In Proc. of the *International Requirements Engineering Conference (RE '07)*. IEEE Computer Society, November 2007.

[46] N. S. Godbole. *Software quality assurance: Principles and practice*. Alpha Science, Oxford, 2. repr. edition, 2007.

[47] A. Goeb and K. Lochmann. A software quality model for SOA. In Proc. of the *International Workshop on Software Quality (WoSQ '11)*. ACM, September 2011.

[48] R. B. Grady and D. L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Prentice Hall, 1987.

[49] P. Grubb and A. A. Takang. *Software maintenance: Concepts and practice*. World Scientific, New Jersey NJ u.a, 2 edition, 2003.

[50] H. Gruber. Benchmarking-oriented Assessment of Source Code Quality, PhD thesis, Johannes Kepler Universität. 2010.

[51] H. Gruber, R. Plösch, and M. Saft. On the validity of benchmarking for evaluating code quality. In Proc. of the *International Conferences on Software Measurement (IWSM/MetriKon/-Mensura '10)*. Shaker, 2010.

[52] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, 2005.

[53] P. Hájek. *Metamathematics of fuzzy logic*, volume 4 of *Trends in logic*. Kluwer, Dordrecht u.a, 1998.

[54] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A Systematic Review of Fault Prediction Performance in Software Engineering. *IEEE Transactions on Software Engineering*, PP(99):1, 2011.

[55] M. H. Halstead. *Elements of Software Science*. Operating and programming systems series. Elsevier Science Inc, New York, NY, USA, 1977.

[56] D. Harel and B. Rumpe. *Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff, Technical Report of The Weizmann Institute of Science, Rehovot, Israel*. Number MCS00-16. 2000.

[57] L. Harvey and D. Green. Defining Quality. *Assessment & Evaluation in Higher Education*, 18(1):9–34, 1993.

[58] I. Heitlager, T. Kuipers, and J. Visser. A Practical Model for Measuring Maintainability. In Proc. of the *International Conference on the Quality of Information and Communication Technology (QUATIC '07)*. IEEE Computer Society, September 2007.

[59] D. W. Hoffmann. *Software-Qualität*. eXamen.press. Springer, Berlin u.a, 2008.

[60] F. Hutton Barron. Selecting a best multiattribute alternative with partial information about attribute weights. *Acta Psychologica*, 80(1-3):91–103, 1992.

[61] IEEE. Std 610.12-1990, Standard Glossary of Software Engineering Terminology, 1990.

[62] IEEE. Std 1219-1998, Standard for Software Maintenance, 1998.

[63] IEEE. Std 1074-2006, Standard for Developing a Software Project Life Cycle Process, 2006.

[64] IEEE. Std 12207, Standard for Systems and Software Engineering - Software Life Cycle Processes: IEEE STD 12207-2008, 2008.

[65] ISO. 25010, Systems and software engineering – System and software product Quality Requirements and Evaluation (SQuaRE) – System and software quality models.

[66] ISO. 25020, Systems and software engineering – System and software product Quality Requirements and Evaluation (SQuaRE) –Measurement reference model and guide.

[67] ISO. 25021, Systems and software engineering – System and software product Quality Requirements and Evaluation (SQuaRE) – Quality measure elements.

[68] ISO. 9000, Quality management systems - Fundamentals and vocabulary.

[69] ISO. 9241, Ergonomics of human-system interaction.

[70] ISO. 9126-1, Software engineering - Product quality - Part 1: Qualiy model, 1991.

[71] ISO. 15504, Software Process Improvement and Capability Determination (SPICE), 2007.

[72] ISO. 26262, Road vehicles – Functional safety, 2011.

[73] I. Jacobson. The use-case construct in object-oriented software engineering. pages 309–336, 1995.

[74] M. Jarke, X. T. Bui, and J. M. Carroll. Scenario Management: An Interdisciplinary Approach. *Requirements Engineering Journal*, 3(3):155–173, 1998/03/20/.

[75] JavaDepend. http://www.javadepend.com, last accessed on 11.07.2012, 2012.

[76] C. Jones. *Software assessments, benchmarks, and best practices*. Addison-Wesley information technology series. Addison-Wesley, Boston Mass. u.a, 1. print. edition, 2000.

[77] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit. Can clone detection support quality assessments of requirements specifications? In Proc. of the *International Conference on Software Engineering (ICSE '10)*, volume 2. ACM, May 2010.

[78] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In Proc. of the *International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, May 2009.

[79] J. M. Juran. *Juran's quality handbook*. McGraw-Hill, New York NY u.a., 5. ed., internat. ed. edition, 2000.

[80] H. Kaiya, H. Horai, and M. Saeki. AGORA: Attributed Goal-Oriented Requirements Analysis Method. In Proc. of the *International Conference on Requirements Engineering (RE '98)*. 1998.

[81] R. Kazman, m. Klein, and P. Clements. *ATAM: Method for Architecture Evaluation: Technical Report of Carnegie-Mellon University, Pittsburgh, PA, Software Engineering Institure*. Number ADA382629. 2000.

[82] T. M. Khoshgoftaar and N. Seliya. Fault Prediction Modeling for Software Quality Estimation: Comparing Commonly Used Techniques. *Empirical Software Engineering*, 8(3):255–283, 2003.

[83] C. W. Kirkwood and R. K. Sarin. Ranking with Partial Information: A Method and an Application. *Operations Research*, 33(1):38–48, 1985.

[84] B. Kitchenham. Software quality assurance. *Microprocessors and Microsystems*, 13(6):373–381, 1989.

[85] B. Kitchenham, S. Linkman, A. Pasquini, and V. Nanni. The SQUID approach to defining a quality model. *Software Quality Journal*, 6(3):211–233, 1997.

[86] B. Kitchenham and S. L. Pfleeger. Software Quality: The Elusive Target. *IEEE Software*, 13(1):12–21, 1996.

[87] M. Kläs, J. Heidrich, J. Muench, and A. Trendowicz. CQML Scheme: A Classification Scheme for Comprehensive Quality Model Landscapes. In Proc. of the *Euromico Conference on Software Engineering and Advanced Applications (SEAA '09)*. IEEE Computer Society, August 2009.

[88] M. Kläs, C. Lampasona, S. Nunnenmacher, S. Wagner, M. Herrmannsdoerfer, and K. Lochmann. How to Evaluate Meta-Models for Software Quality? In Proc. of the *International Conferences on Software Measurement (IWSM/MetriKon/Mensura '10)*. Shaker, 2010.

[89] M. Kläs, K. Lochmann, and L. Heinemann. Evaluating a Quality Model for Software Product Assessments - A Case Study. In Proc. of the *Workshop zur Software-Qualitätsmodellierung und -bewertung (SQMB '11), Technical Report of Technische Universität München*. Februar 2011.

[90] G. Kotonya and I. Sommerville. *Requirements engineering*. Worldwide series in computer science. Wiley, Chichester [u.a.], 2000.

[91] P. Kruchten. *The rational unified process: An introduction*. The Addison-Wesley object technology series. Addison-Wesley, Upper Saddle River NJ u.a, 3. ed., 6. pr. edition, 2007.

[92] J. Kumar Chhabra, K. Aggarwal, and Y. Singh. Code and data spatial complexity: two important software understandability measures. *Information and Software Technology*, 45(8):539–546, 2003.

[93] K. Kvam. XRadar, http://xradar.sourceforge.net/, last accessed on 11.07.2012, 2012.

[94] M. Lanza and R. Marinescu. *Object-oriented metrics in practice: Using software metrics to characterize, evaluate, and improve the design of object-oriented systems*. Springer, Berlin u.a, 2006.

[95] E. Letier. Reasoning about Agents in Goal-Oriented Requirements Engineering: PdD Thesis, 2001.

[96] LimeSurvey. version 1.81+, http://www.limesurvey.org/, last accessed on 11.07.2012.

[97] R. Lincke, T. Gutzmann, and W. Löwe. Software Quality Prediction Models Compared. In Proc. of the *International Conference on Quality Software (QSIC 2010)*. 2010.

[98] Z. Liu, H. Jifeng, X. Li, and Y. Chen. A Relational Model for Formal Object-Oriented Requirement Analysis in UML. In Proc. of the *Formal Methods and Software Engineering*, volume 2885 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003.

[99] K. Lochmann. Engineering Quality Requirements Using Quality Models. In Proc. of the *International Conference on Engineering of Complex Computer Systems (ICECCS' 10)*. IEEE Computer Society, March 2010.

[100] K. Lochmann. A Benchmarking-inspired Approach to Determine Threshold Values for Metrics. In Proc. of the *Workshop on Software Quality (WoSQ '12)*. ACM, November 2012.

[101] K. Lochmann and A. Goeb. A Unifying Model for Software Quality. In Proc. of the *International Workshop on Software Quality (WoSQ '11)*. ACM, September 2011.

[102] K. Lochmann and L. Heinemann. Integrating Quality Models and Static Analysis for Comprehensive Quality Assessment. In Proc. of the *International Workshop on Emerging Trends in Software Metrics (WETSoM '11)*. ACM, May 2011.

[103] K. Lochmann, D. Mendez Fernandez, and S. Wagner. A Case Study on Specifying Quality Requirements Using a Quality Model. In Proc. of the *Asia Pacific Software Engineering Conference (APSEC '12)*. IEEE Computer Society, December 2012.

[104] K. Lochmann, S. Wagner, A. Goeb, and D. Kirchler. Classification of Software Quality Patterns. In Proc. of the *Workshop zur Software-Qualitätsmodellierung und -bewertung (SQMB '10), Technical Report of Technische Universität München, TUM-I1001*. February 2010.

[105] M. Luckey, A. Baumann, D. Mendez Fernandez, and S. Wagner. Reusing security requirements using an extended quality model. In Proc. of the *Workshop on Software Engineering for Secure Systems (SESS '10)*. ACM, May 2010.

[106] C. Marinescu, R. Marinescu, R. F. Mihancea, D. Ratiu, and R. Wettel. iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design. In Proc. of the *International Conference on Software Maintenance (ICSM '05)*. IEEE Computer Society, 2005.

[107] F. J. J. Massey. The Kolmogorov-Smirnov Test for Goodness of Fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.

[108] T. J. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976.

[109] J. A. McCall, P. K. Richards, and G. F. Walters. *Factors in Software Quality*. NTIS, 1977.

[110] D. Mendez Fernandez and M. Kuhrmann. *Artefact-based Requirements Engineering and its Integration into a Process Framework: A Customisable Model-based Approach for Business Information Systems' Analysis: Technical Report of Technische Universität München*. Number TUM-I0929. 2009.

[111] Mono Project. Gendarme, http://www.mono-project.com/Gendarme, last accessed on 11.07.2012, 2011.

[112] K. Mordal, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse. Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process*, 2012.

[113] K. Mordal-Manet, F. Balmas, S. Denier, S. Ducasse, H. Wertz, J. Laval, F. Bellingard, and P. Vaillergues. The Squale Model – A Practice-based Industrial Quality Model. In Proc. of the *International Conference on Software Maintenance (ICSM '09)*. IEEE Computer Society, September 2009.

[114] Motor Industry Software Reliability Association. *MISRA-C:2004: Guidelines for the use of the C language in critical systems*. MIRA, Nuneaton, second edition, 2004.

[115] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In Proc. of the *International Conference on Software Engineering (ICSE '05)*. ACM, May 2005.

[116] Object Management Group, Inc. Unified Modeling Language 2.0, http://www.omg.org/spec/UML/2.0/, last accessed on 11.07.2012. 2005.

[117] M. Ortega, M. Pérez, and T. Rojas. Construction of a Systemic Quality Model for Evaluating a Software Product. *Software Quality Journal*, 11(3):219–242, 2003.

[118] K. Oshiro, K. Watahiki, and M. Saeki. Goal-oriented idea generation method for requirements elicitation. In Proc. of the *International Conference on Requirements Engineering (RE'03)*. IEEE Computer Society, 2003.

[119] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes. SourcererDB: An aggregated repository of statically analyzed and cross-linked open source Java projects. In Proc. of the *International Working Conference on Mining Software Repositories (MSR '09)*. IEEE Computer Society, June 2009.

[120] PC-lint. 9.00, http://www.gimpel.com/, last accessed on 11.07.2012, 2011.

[121] H. Pham. *Software reliability*. Springer, Singapore and u.a, 2000.

[122] R. Plösch, H. Gruber, A. Hentschel, C. Körner, G. Pomberger, S. Schiffer, M. Saft, and S. Storck. The EMISQ method and its tool support-expert-based evaluation of internal software quality. *Innovations in Systems and Software Engineering*, 4(1):3–15, 2008.

[123] R. Plosch, H. Gruber, A. Hentschel, C. Körner, G. Pomberger, S. Schiffer, M. Saft, and S. Storck. The EMISQ Method - Expert Based Evaluation of Internal Software Quality. In Proc. of the *Software Engineering Workshop (SEW '07)*. IEEE Computer Society, March 2007.

[124] R. Plösch, H. Gruber, C. Körner, and M. Saft. A Method for Continuous Code Quality Management Using Static Analysis. In Proc. of the *International Conference on the Quality of Information and Communications Technology (QUATIC '10)*. IEEE Computer Society, September 2010.

[125] PMD. http://pmd.sourceforge.net/, last accessed on 11.07.2012, 2011.

[126] Quamoco Base Model. http://www.quamoco.de/tools, last accessed on 11.07.2012, 2012.

[127] A. Rainer and S. Gale. Evaluating the Quality and Quantity of Data on Open Source Software Projects. In Proc. of the *International Conference on Open Source Systems (ICOSS '05)*. UHRA, June 2005.

[128] G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting Its Multiview Approach. In Proc. of the *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2001.

[129] Reinhold Ploesch, Harald Gruber, Gustav Pomberger, Matthias Saft, and Stefan Schiffer. Tool Support for Expert-Centred Code Assessments. In Proc. of the *International Conference on Software Testing, Verification, and Validation (ICST '08)*. IEEE Computer Society, June 2008.

[130] R. Roberts and P. Goodwin. Weight approximations in multi-attribute decision models. *Journal of Multi-Criteria Decision Analysis*, 11(6):291–303, 2002.

[131] S. Robertson and J. Robertson. *Mastering the requirements process*. Addison-Wesley, Upper Saddle River NJ, 2nd ed. edition, 2006.

[132] J. Rumbaugh, I. Jacobson, and G. Booch. *The unified modeling language reference manual: The definitive reference to the UML from the original designers*. 1. edition, 1999.

[133] I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos. The SQO-OSS quality model: Measurement based open source software evaluation. In Proc. of the *International Conference on Open Source Systems (ICOSS '08)*. Springer, 2008.

[134] H. Schackmann, M. Jansen, and H. Lichter. Tool Support for User-Defined Quality Assessment Models. In Proc. of the *International Conferences on Software Measurement (IWS-M/MetriKon/Mensura '09)*. Shaker, November 2009.

[135] P. J. H. Schoemaker and C. C. Waid. An Experimental Comparison of Different Approaches to Determining Weights in Additive Utility Models. *Management Science*, 28(2):182–196, 1982.

[136] S. S. Shapiro and M. B. Wilk. An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 52(3/4):591–611, 1965.

[137] G. Sindre and A. L. Opdahl. Eliciting security requirements with misuse cases. *Requirements Engineering Journal*, 10(1):34–44, 2005/01/01/.

[138] Software Engineering Institute. Capability Maturity Model Integration (CMMI), http://www.sei.cmu.edu/cmmi/, last accessed on 11.07.2012, 2012.

[139] I. Sommerville and P. Sawyer. *Requirements engineering: A good practice guide*. Wiley, Chichester, reprinted. edition, 2006.

[140] Sonar. http://www.sonarsource.org/, last accessed on 11.07.2012, 2010.

[141] C. Spearman. The Proof and Measurement of Association between Two Things. *The American Journal of Psychology*, 15(1):72–101, 1904.

[142] Squale Project. The Squale Quality Model, http://www.squale.org/quality-models-site/research-deliverables/WP1.3_Practices-in-the-Squale-Quality-Model_v2.pdf, last accessed on 11.07.2012. 2010.

[143] S. S. Stevens. On the Theory of Scales of Measurement. *Science of Computer Programming*, 103(2684):677–680, 1946.

[144] W. G. Stillwell, D. A. Seaver, and W. Edwards. A comparison of weight approximation techniques in multiattribute utility decision making. *Organizational Behavior and Human Performance*, 28(1):62–77, 1981.

[145] Swoyer and Chris. Properties (Stanford Encyclopedia of Philosophy), http://plato.stanford.edu/entries/properties/, last accessed on 11.07.2012, 2011.

[146] The Eclipse Foundation. Eclipse Project, http://www.eclipse.org/, last accessed on 11.07.2012, 2012.

[147] D. H. Traeger. *Einführung in die Fuzzy-Logik*. Teubner, Stuttgart, 1993.

[148] M. D. Troutt and W. Acar. A Lorenz-Pareto measure of pure diversification. *European Journal of Operational Research*, 167(2):543–549, 2005.

[149] Understand. http://www.scitools.com/, last accessed on 11.07.2012, 2012.

[150] V-Modell ® XT. Version 1.3, http://v-modell.iabg.de/dmdocuments/V-Modell-XT-Gesamt-Englisch-V1.3.pdf, last accessed on 11.07.2012. 09.02.2009.

[151] A. van Lamsweerde. *Requirements engineering: From system goals to UML models and software specifications*. Wiley, Chichester, 2009.

[152] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *IEEE Transactions on Software Engineering*, 24(11):908–926, 1998.

[153] F. van Latum, R. van Solingen, M. Oivo, B. Hoisl, D. Rombach, and G. Ruhe. Adopting GQM based measurement in an industrial environment. *IEEE Software*, 15(1):78–86, 1998.

[154] R. van Solingen and E. Berghout. Integrating goal-oriented measurement in industrial software engineering: industrial experiences with and additions to the Goal/Question/Metric method (GQM). In Proc. of the *International Software Metrics Symposium (METRICS '01)*. IEEE Computer Society, April 2001.

[155] R. H. J. van Zeist and P. R. H. Hendriks. Specifying software quality with the extended ISO model. *Software Quality Journal*, 5(4):273–284, 1996.

[156] S. Wagner, F. Deissenboeck, and S. Winter. Erfassung, Strukturierung und überprüfung von Qualitätsanforderungen durch aktivitätenbasierte Qualitätsmodelle. In Proc. of the *Software Engineering Konferenz (SE '08)*. GI, February 2008.

[157] S. Wagner, F. Deissenboeck, and S. Winter. Managing quality requirements using activity-based quality models. In Proc. of the *International Workshop on Software Quality (WoSQ '08)*. ACM, May 2008.

[158] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, C. Lampasona, A. Trendowicz, R. Plösch, A. Mayr, A. Seidl, A. Goeb, and J. Streit. Practical Product Quality Modelling and Assessment: The Quamoco Appraoch (under review, submitted in May 2012). *IEEE Transactions on Software Engineering*, 2012.

[159] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, A. Trendowicz, R. Plösch, A. Seidl, A. Goeb, and J. Streit. The Quamoco Product Quality Modelling and Assessment Approach. In Proc. of the *International Conference on Software Engineering (ICSE '12)*. ACM, June 2012.

[160] S. Wagner, K. Lochmann, S. Winter, F. Deissenboeck, E. Juergens, M. Herrmannsdoerfer, L. Heinemann, M. Kläs, A. Tendowicz, J. Heidrich, R. Ploesch, A. Goeb, C. Koerner, K. Schoder, J. Streit, and C. Schubert. *The Quamoco Quality Meta-Model: Technical Report of Techniche Universität München*. Number TUM-I128. 2012.

[161] S. Wagner, K. Lochmann, S. Winter, A. Goeb, and M. Kläs. Quality Models in Practice: A Preliminary Analysis. In Proc. of the *International Symposium on Empirical Software Engineering and Measurement (ESEM '09)*. IEEE Computer Society, October 2009.

[162] S. Wagner, K. Lochmann, S. Winter, A. Goeb, M. Kläs, and S. Nunnenmacher. *Software Quality Models in Practice – Survey Results: Technical Report of Techniche Universität München*. Number TUM-I129. 2012.

[163] S. Wagner, D. Mendez Fernandez, S. Islam, and K. Lochmann. A Security Requirements Approach for Web Systems. In Proc. of the *Workshop Quality Assessment in Web (QAW '09)*. Springer, June 2009.

[164] E. Wallmüller. *Software-Qualitätssicherung in der Praxis*. Hanser, München, 1990.

[165] M. Weber and K. Borcherding. Behavioral influences on weight judgments in multiattribute decision making. *European Journal of Operational Research*, 67(1):1–12, 1993.

[166] S. Winter, S. Wagner, and F. Deissenboeck. A Comprehensive Model of Usability. *Engineering Interactive Systems*, 2008(4940):106–122, 2008.

[167] C. Wohlin, P. Runeson, M. Höst, Magnus C. Ohlsson and Björn Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Springer, 2000.

[168] World Wide Web Consortium. Web Content Accessibility Guidelines (WCAG) 2.0, http://www.w3.org/TR/2008/REC-WCAG20-20081211/, last accessed on 11.07.2012, 2008.

[169] B. Xhenseval. QALab, http://qalab.sourceforge.net/, last accessed on 11.07.2012.

[170] S. H. Zanakis, A. Solomon, N. Wishart, and S. Dublish. Multi-attribute decision making: A simulation comparison of select methods. *European Journal of Operational Research*, 107(3):507–529, 1998.

A Appendix

A.1 Fuzzy Propositional Logic

A.1.1 Definition of the Łukasiewicz Propositional Logic

We define Łukasiewicz propositional logic according to [53, p. 63] and [15, p. 209]. The syntax of fuzzy propositional logic Ł has propositional variables $p_1, p_2, \ldots$, operators $\otimes$, $\wedge$, $\vee$, $\rightarrow$, and $\neg$, the constants 0 and 1. Formulas are defined as follows: Each propositional variable is a formula, each constant is a formula; if ϕ and γ are formulas, then (ϕ), $\neg\phi$, $(\phi \wedge \gamma)$, $(\phi \vee \gamma)$, and $(\phi \rightarrow \gamma)$ are formulas. For the semantics of Ł we define: Let Ω be the set of all formulas, then we define the σ-operator that assigns a truth value to each formula: $\sigma : \Omega \rightarrow [0, 1]$.

For each formula ϕ with variables $p_1, \ldots, p_n$ (written as $\phi(p_1, \ldots, p_n)$) the σ-operator is defined as:

$$\sigma(\phi(p_1, \ldots, p_n)) = \phi(\sigma(p_1), \ldots, \sigma(p_n)) \tag{A.1}$$

For the constants we define:

$$\begin{aligned} \sigma(0) &= 0 \\ \sigma(1) &= 1 \end{aligned} \tag{A.2}$$

For two formulas ϕ and γ the operators are defined as:

$$\begin{aligned} \sigma(\neg_Ł \phi) &= 1 - \sigma(\phi) \\ \sigma(\phi \otimes_Ł \gamma) &= max(0, \sigma(\phi) + \sigma(\gamma) - 1) \\ \sigma(\phi \wedge_Ł \gamma) &= min(\sigma(\phi), \sigma(\gamma)) \\ \sigma(\phi \vee_Ł \gamma) &= max(\sigma(\phi), \sigma(\gamma)) \\ \sigma(\phi \rightarrow_Ł \gamma) &= min(1, 1 - \sigma(\phi) + \sigma(\gamma)) \end{aligned} \tag{A.3}$$

An n-ary formula $\phi(p_1, \ldots, p_n)$ is a tautology if

$$\sigma(\phi(p_1, \ldots, p_n)) = 1 \tag{A.4}$$

for all $(\sigma(p_1), \ldots, \sigma(p_n))$.

A.2 Fuzzy Operators and Modifiers

We collect binary fuzzy operators from literature:

Operation	Source	Definition
Ł strong comp.	[53, p. 63]	$a \otimes_{\text{Ł}} b = max(0, a + b - 1)$
Goedel strong comp.	[53, p. 97]	$a \otimes_{Goedel} b = min(a, b)$
Product strong comp.		$a \otimes_{Product} b = a \cdot b$
Ł, Goedel, Product weak composition	[53, p. 36]	$a \wedge_{\text{Ł}} b = min(a, b)$
Ł, Goedel, Product weak disjunction	[53, p. 36]	$a \vee_{\text{Ł}} b = max(a, b)$
Probabilistic sum		$a \vee_{probabilistic} b = a + b - a * b$
Bounded sum		$a \vee_{boundedsum} b = min(1, a + b)$
Ł neg.	[53, p. 63]	$\neg_{\text{Ł}} a = 1 - a$
Goedel neg.		$\neg_{Goedel} a = \begin{cases} 1 & \text{if } a = 0 \\ 0 & \text{if } a \neq 0 \end{cases}$
Zadeh imp.	[15, p. 266]	$a \rightarrow_{zadeh} b = max(min(x, y), 1 - x)$
Mamdani imp.	[15, p. 266]	$a \rightarrow_{mamdani} b = min(a, b)$
Ł imp.	[15, p. 266]	$a \rightarrow_{\text{Ł}} b = min(1, 1 - a + b)$
Goedel imp.	[15, p. 266]	$a \rightarrow_{goedel} b = \begin{cases} 1 & \text{if } x \leq y \\ y & \text{otherwise} \end{cases}$
Kleene-Dienes imp.	[15, p. 266]	$a \rightarrow_{kleene-dienes} b = max(1 - x, y)$
Goguen imp.	[15, p. 266]	$a \rightarrow_{goguen} b = \begin{cases} 1 & \text{if } x = 0 \\ min(1, y/x) & \text{otherwise} \end{cases}$
Gaines-Rescher imp.	[15, p. 266]	$a \rightarrow_{gaines-rescher} b = \begin{cases} 1 & \text{if } x \leq y \\ 0 & \text{otherwise} \end{cases}$
Reichenbach imp.	[15, p. 266]	$a \rightarrow_{reichenbach} b = 1 - x + x * y$
Larsen imp.	[15, p. 266]	$a \rightarrow_{larsen} b = x * y$

We collect n-ary fuzzy operators from literature:

Operator
Weighted Sum Composition $\wedgeonwedge(x_1, \dots, x_n) = \sum w_i * x_i$ $w_i > 0$ $\sum w_i = 1$
Compensatory Lambda-Operator [147, p. 42] $a\lambda b = \lambda * a * b + (1 - \lambda) * (a + b - a * b)$ $\lambda \in [0, 1]$ $\lambda = 0 \Rightarrow a\lambda b = a + b - a * b = a \vee_{probabilistic} b$ $\lambda = 1 \Rightarrow a\lambda b = a * b = a \otimes_{Product} b$
Compensatory Gamma-Operator [147, p. 43] $a\gamma b = (a * b)^{(1-\gamma)} * (1 - (1 - a) * (1 - b))^{(\gamma)}$ $\gamma \in [0, 1]$ $\gamma = 0 \Rightarrow a * b = a \otimes_{Product} b$ $\gamma = 1 \Rightarrow a * b = (1 - (1 - a) * (1 - b)) = a \vee_{probabilistic} b$
Compensatory Weighted Gamma-Operator n-ary [147, p. 45] $\gamma(x_1, \dots, x_n) = \left(\prod_{i=1}^{n} x_i^{w_i}\right)^{1-\gamma} * (1 - \prod_{i=1}^{n} (1 - x_i)^{w_i})^{\gamma}$ $\gamma \in [0, 1]$ $\sum w_i = 1$
Powering-Modifier [147, p. 48] $\alpha(x) = x^{\alpha}$
Gamma-Modifier (modification of Gamma-Operator) $\alpha(x) = x^{1-\alpha} * (1 - (1 - x)^{\alpha})$
Exp-Modifier $\alpha(x) = \frac{\alpha^x - 1}{\alpha - 1}$