Chapter 2
Quality Models

In this chapter, after discussing existing quality models and putting them into context, I will introduce basics of software measures and details of the ISO/IEC 25010 quality model. The main part of this chapter constitutes the quality modelling approach developed in the research project Quamoco, how to maintain such quality models and three detailed examples of quality models.

2.1 Quality Models Set into Context

This whole chapter is dedicated to software quality models, because they constitute the foundation of software product quality control as we understand it in this book. There is a lot of existing literature and approaches for software quality models. Therefore, we start with setting our notion of quality models into context with a short history, definitions and classifications and a description of an example for each class of quality models.

2.1.1 A Short History of Software Quality Models

Quality models have been a research topic for several decades and a large number of quality models have been proposed [125]. We cannot cover the area completely, but we will look into three historical threats in the development of quality models: hierarchical models, meta-model-based models and implicit quality models.

Hierarchical Quality Models

The first published quality models for software date back to the late 1970s, when Boehm et al. [24] as well as McCall, Richards and Walter [149] described quality

S. Wagner, *Software Product Quality Control*, DOI 10.1007/978-3-642-38571-1_2,

characteristics and their decomposition. The two approaches are similar and use a hierarchical decomposition of the concept *quality* into quality factors such as *maintainability* or *reliability*. Several variations of these models have appeared over time. One of the more popular ones is the FURPS model [77] which decomposes quality into functionality, usability, reliability, performance and supportability. Besides this hierarchical decomposition, the main idea of these models is that you decompose quality down to a level where you can measure and, thereby, evaluate quality.

This kind of quality models became the basis for the international standard ISO/IEC 9126 [107] in 1991. The standard defines a standard decomposition into quality characteristics and suggests a small number of metrics for measuring them. These metrics do not cover all aspects of quality, however. Hence, the standard does not completely operationalise quality. The successor of ISO/IEC 9126, the new standard ISO/IEC 25010 [97], changes a few classifications but keeps the general hierarchical decomposition.

In several proposals, researchers have used metrics to directly measure quality characteristics from or similar to ISO/IEC 9126. Franch and Carvallo [66] adapt the ISO quality model and assign metrics to measure them for selecting software packages. They stress that they need to be able to explicitly describe "relationships between quality entities". Van Zeist and Hendriks [219] also extend the ISO model and attach measures such as *average learning time*. Samoladas et al. [185] use several of the quality characteristics of ISO/IEC 9126 and extend and adapt them to open source software. They use the quality characteristics to aggregate measurements to an ordinal scale. All these approaches reveal that it is necessary to extend and adapt the ISO standard. They also show the difficulty in measuring abstract quality characteristics directly.

Ortega, Perez and Rojas [169] take a notably different view by building what they call a *systemic quality model* and using *product effectiveness* and *product efficiency* as the basic dimensions to structure similar quality characteristics as in the other approaches, such as reliability or maintainability.

Experimental research tools (e.g. [142, 187]) take first steps towards integrating a quality model and an assessment toolkit. A more comprehensive approach is taken by the research project Squale [155]. Here, an explicit quality model is developed that describes a hierarchical decomposition of the ISO/IEC 9126 quality characteristics. The model contains formulas for aggregating and normalising metric values. Based on the quality model, Squale provides tool support for evaluating software products. The measurements and the quality model are fixed within the tools.

Various critiques (e.g. [2,51]) point out that the decomposition principles used for quality characteristics are often ambiguous. Furthermore, the resulting quality characteristics are mostly not specific enough to be measurable directly. Although the recently published successor ISO/IEC 25010 has several improvements, including a measurement reference model in ISO/IEC 25020, the overall critique is still valid because detailed measures are yet missing. Moreover, a survey done by us [213]

shows that less than 28 % of the companies use these standard models and 71 % of them have developed their own variants. Hence, there is a need for customisation.

Meta-Model-Based Quality Models

Starting in the 1990s, researchers have been proposing more elaborate ways of decomposing quality characteristics and thereby have built richer quality models based on more or less explicit meta-models. A meta-model, similar as in the UML,[1] is a model of the quality model, i.e. it describes how valid quality models are structured. In addition, the proposals also differ to what extent they include measurements and evaluations.

An early attempt to make a clear connection between measurements and the quality factors described in the hierarchical quality models was made in the ESPRIT project *REQUEST*. They developed what they called a *constructive quality model* COQUAMO [120, 124]. They see the quality factor as central in a quality model and argue that each factor should be evaluated differently throughout different development phases and, therefore, have different metrics. They also differentiate between application-specific and general qualities. For example, they see reliability as important for any software system, but security is application specific. COQUAMO aimed strongly at establishing quantitative relationships between quality drivers measured by metrics and factors. We will discuss COQUAMO in more detail below as an example for multi-purpose models.

With some similarity is the also similarly named COQUALMO [37]. It uses drivers, similar to the effort model COCOMO [23], and describes a defect flow model. In each phase, defects are introduced by building and changing artefacts and removed by quality assurance techniques. For example, requirements defects are introduced because of misunderstood expectations of the stakeholders. Some of these requirements defects are then removed by reviews or later testing. An estimation of the drivers enables COQUALMO to estimate the defect output.

Without an explicit meta-model but with a similar focus on establishing statistical relationships, there is a plethora of various very specific quality models. They use a set of measure which are expected to influence a specific quality factor. For example, the *maintainability index* (MI) [41] was a set of static code measures combined so that the resulting index should give an indication of the maintainability of the system. Another example is reliability growth models [139] which relate test data with the reliability of the software.

Dromey [54] published a quality model with a comparably elaborate meta-model in which he distinguishes between *product components*, which exhibit *quality-carrying properties*, and externally visible *quality attributes*. Product components are parts of the product, in particular the source code of the software. For example, a variable is a product component with quality-carrying properties such as *precise*,

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

assigned or *documented*. These are then set into relation to the quality factors of ISO/IEC 9126.

Kitchenham et al. [21, 121] built the SQUID approach on the experiences from COQUAMO. They acknowledge the need for an explicit meta-model to describe the increasingly complex structures of quality models. Although they state in [121] that the REQUEST project concluded that "there were no software product metrics that were, in general, likely to be good predictors of final product qualities" and that there is still no evidence on that, it is still useful to model and analyse influences to quality. Hence, they propose to monitor and control "internal" measures which may influence the "external" quality. Their quality meta-model includes *measurable properties* that can be *internal software properties* or *quality subcharacteristics*. The internal software properties influence the quality subcharacteristics and both can be measured. They also introduce the notion to define a target value for the measurable properties for a quality requirements specification.

Bansiya and Davis [7] built on Dromey's model and proposed QMOOD, a quality model for object-oriented designs. They described several metrics for the design of components to measure what they call *design properties*. These properties have an influence on quality attributes. They also explicitly mentioned tool support and describe empirical validations of their model.

Bakota et al. [6] emphasised the probabilistic nature of their quality model and quality assessments. They introduced *virtual quality attributes* which are similar to the internal software properties of SQUID. The quality model uses only nine low-level measures which are evaluated and aggregated to probability distributions. Our experience has been, however, that practitioners have difficulties interpreting such distributions.

All these meta-model-based quality models show that the complex concept of *quality* needs more structure in quality models than abstract quality characteristics and metrics. They have not established a general base quality model, however, which you can just download and apply.

Statistical and Implicit Quality Models

For various quality factors, statistical models have been proposed that capture properties of the product, process, or organisation and estimate or predict these quality factors. A prominent example of such models are *reliability growth models* [139,158]. They transfer the idea of hardware reliability models to software. The idea is to observe the failure behaviour of a software, for example, during system testing, and predict how this behaviour will change over time. Similar models are the *maintainability index* (MI) [41], a regression model from code metrics or Vulture [166], a machine learning model predicting vulnerable components based on vulnerability databases and version archives.

Although they usually do not explicitly say so, many quality analysis tools use some kind of quality model. For example, tools for bug pattern identification (e.g. FindBugs, Gendarme or PC-Lint) classify their rules into different categories based on the type of problems they detect (dodgy code), what quality factor it might influence (e.g. performance) or how severe the problem is. Therefore, implicitly, these tools already define a lot what we would expect from a quality model: quality factors, measurements and influences. Usually, it is not made explicit, however, which makes a comprehensive quality evaluation difficult.

Dashboards can use the measurement data of these tools as input (e.g. QALab, Sonar or XRadar). Their goal is to present an overview of the quality data of a software system. Nevertheless, they often also lack an explicit connection between the metrics used and the required quality factors. Hence, explanations of the impacts of defects on software quality and rationales for the used metrics are missing.

Finally, also checklists used in development or reviews are a kind of quality model. They are usually not directly integrated with a quality model at all. They define properties of artefacts, most often source code, which have an influence on some quality factor. Sometimes the checklists make these influences explicit. Most often, they do not. This leaves the software engineers without a clear rationale why to follow the guidelines. Hence, explicit quality models can also help us to improve checklists and guidelines.

2.1.2 Definitions and Classifications

Software quality models are now a well-accepted means to support quality control of software systems. As we have seen, over the last three decades a multitude of very diverse models commonly termed "quality models" have been proposed. On first sight, such models appear to have little relation to each other although all of them deal with software quality. We claim that this difference is caused by the different purposes the models pursue [49]: The ISO/IEC 25010 is mainly used to *define* quality, metric-based approaches are used to *assess* the quality of a given system and reliability growth models are used to *predict* quality. To avoid comparing apples with oranges, we propose to use these different purposes, namely, *definition, assessment* and *prediction* of quality, to classify quality models. Consequently, we term the ISO/IEC 25010 as *definition model*, the maintainability index as *assessment model* and reliability growth models as *prediction models*.

Although *definition, assessment* and *prediction* of quality are different purposes, they are not independent of each other: It is hard to assess quality without knowing what it actually constitutes and equally hard to predict quality without knowing how to assess it. This relation between quality models is illustrated by the DAP classification shown in Fig. 2.1.

The DAP classification views prediction models as the most advanced form of quality models as they can also be used for the definition of quality and for its

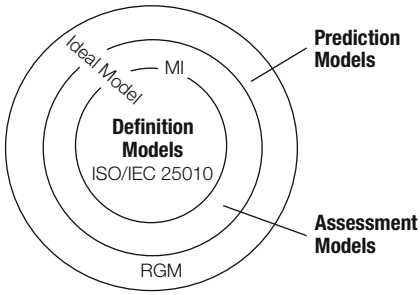

Fig. 2.1 DAP classification for quality models [49]

assessment. This view only applies for ideal models, however. As Fig. 2.1 shows, existing quality models do not necessarily cover all aspects equally well. The ISO/IEC 25010, for example, defines quality but gives no hints for assessing it; the MI defines an assessment whose relation to a definition of quality is unclear. Similarly, reliability growth models perform predictions based on data that is not explicitly linked to an overall definition of quality.

To reflect the importance of the purpose, we propose a definition of quality models [49] that explicitly takes the purpose of the quality model into account. Due to the diversity of different models, we do not restrict the type of model to a specific modelling technique or formalism.

Definition 2.1 (Quality Model). A model with the objective to describe, assess and/or predict quality.

Independent of the modelling technique used to build a quality model, we consider the existence of a defined meta-model as crucial. Even though, in many cases, quality meta-models are not explicitly defined for existing quality models. A meta-model is required to precisely define the model elements and their interactions. It not only defines legal model instances but also explains how models are to be interpreted. Accordingly, we define [49]:

Definition 2.2 (Quality Meta-Model). A model of the constructs and rules needed to build specific quality models.

Finally, it must be defined how the quality model can be used in the development and evolution of a software system. This typically concerns the process by which a quality model is created and maintained and the tools employed for its operationalisation. We call this a *quality modelling framework*.

Definition 2.3 (Quality Modelling Framework). A framework to define, evaluate and improve quality. This usually includes a quality meta-model as well as a methodology that describes how to instantiate the meta-model and use the model instances for defining, assessing, predicting and improving quality.

Most of the remainder of this book will deal with the Quamoco[2] quality modelling framework. To give you a broader view on the existing quality models, we will first look, however, at some other examples for models for the different purposes.

2.1.3 *Definition Models*

Definition models are used in various phases of a software development process. During requirements engineering, they define quality factors and requirements for planned software systems [122] and thus constitute a method to agree with the customer what quality means [122]. During implementation, quality models serve as basis of modelling and coding standards or guidelines [54]. They provide direct recommendations on system implementation and, thus, constitute constructive approaches to achieve high software quality. Furthermore, quality defects that are found during quality assurance are classified using the quality model [54]. Apart from their use during software development, definitional quality models can be used to communicate software quality knowledge during developer training or student education.

As we will look into the ISO/IEC 25010 quality model in more detail in Sect. 2.3, we use the FURPS model [77] as an example here. FURPS stands for functionality, usability, reliability, performance, and supportability. It is a decomposition of software quality. Each of these quality factors is also further decomposed:

- Functionality
  - Feature set
  - Capabilities
  - Generality
  - Security
- Usability
  - Human factors
  - Aesthetics
  - Consistency
  - Documentation
- Reliability
  - Frequency/severity of failure
  - Recoverability
  - Predictability
  - Accuracy
  - Mean time to failure

[2] http://www.quamoco.de/.

- Performance
  - Speed
  - Efficiency
  - Resource consumption
  - Throughput
  - Response time
- Supportability
  - Testability
  - Extensibility
  - Adaptability
  - Maintainability
  - Compatibility
  - Configurability
  - Serviceability
  - Installability
  - Localisability
  - Portability

Hence, FURPS is a hierarchical definition model. The first four quality factors (FURP) are more aimed at the user and operator of the software, while the last quality factor (S) is more targeted at the developers, testers and maintainers. FURPS gives an alternative decomposition to the standard ISO/IEC 25010 which we will discuss in detail in Sect. 2.3. The main aim of FURPS is a decomposition and checklist for quality requirements. A software engineer can go through this list of quality factors and check with the stakeholders to define corresponding qualities. Therefore, it defines quality as basis for requirements. In addition, Grady and Caswell [77] describe various metrics that can be related to the quality factors for evaluating them. The main purpose of FURPS, however, is to *define* quality.

2.1.4 Assessment Models

Assessment models often extend quality definition models to evaluate the defined qualities of the definition model. During requirements engineering, assessment models can be used to objectively specify and control stated quality requirements [122]. During implementation, the quality model can be the basis for all quality measurements, i.e. for measuring the product, activities and the environment [54, 197, 202]. This includes deriving guidelines for manual reviews [51] and systematically developing and using static analysis tools [54, 174]. Thereby, we monitor and control internal measures that might influence external properties [122]. Apart from their use during software development, assessment models furthermore constitute the touchstone for quality certifications.

The *EMISQ* [174,176] method constitutes an assessment model based on the ISO standard 14598 for product evaluation [92]. It defines an approach for assessing "internal" quality attributes like maintainability and explicitly takes into account the expertise of a human assessor. The method can be used as is with a reference model that is a slight variation of the ISO/IEC 9126 model or customisations thereof. Consequently, the method's quality definition model is very similar to ISO/IEC 9126. It defines quality characteristics and exactly one level of subcharacteristics that are mapped to quality metrics, whereas one subcharacteristic can be mapped to multiple metrics, and vice versa. It uses as metrics results from well-known quality assessment tools like *PC-Lint*[3] and *PMD*.[4] Hence, they include not only classic numeric metrics but also metrics that detect certain coding anomalies. A notable property of the EMISQ method is that its reference model includes about 1,500 different metrics that are mapped to the respective quality characteristics. The approach also provides tool support for building customised quality models and for supporting the assessments.

2.1.5 Prediction Models

In the context of software quality, predictive models have (among others) been applied to predict the number of defects of a system or specific modules, mean times between failures, repair times and maintenance efforts. The predictions are usually based on source code metrics or past defect detection data.

A prediction of reliability is provided by *reliability growth models (RGMs)* that employ defect detection data from test and operation phases to predict the future reliability of software systems [139, 158]. Such models assume that the number of defects found per time unit decreases over time, i.e. the software becomes more stable. The idea then is that if we measure the failure times during system tests with an execution similar to future operation, we will be able to interpolate from them to the failure behaviour in the field. Figure 2.2 illustrates this example data. It shows the calendar time on the x-axis and the cumulated number of failures on the y-axis. Each occurred failure is shown as a cross in the diagram. The curved line is then a fitted statistical model of the failure data. This model goes beyond the already occurred failures and is, hence, able to predict the probable future occurrence of failure. This can then be expressed as reliability.

There are various difficulties in applying reliability growth models. First of all, we need to decide on a suitable statistical model that adequately represents the actual failure distribution. There are many different proposal and it is hard to decide beforehand which one is the best fit. In addition, time measurement is a problem because software does not fail just because clock time passes, but it has to be used.

[3] http://www.gimpel.com.

[4] http://pmd.sourceforge.net.

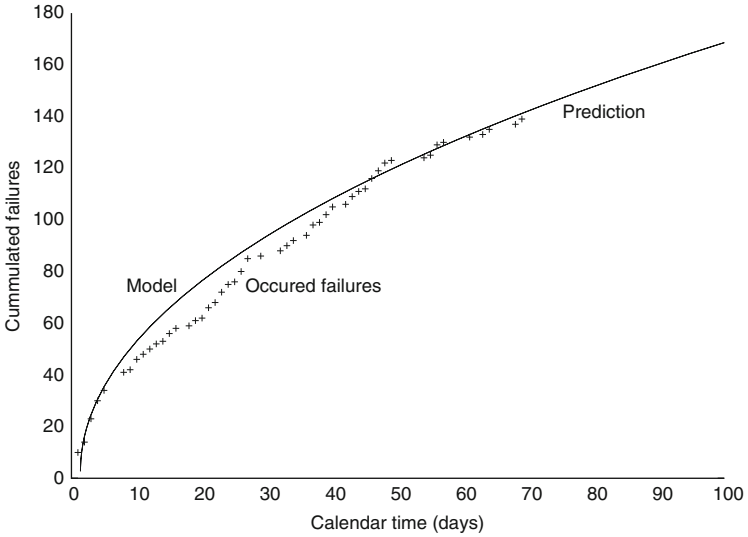

Fig. 2.2 An example prediction of a reliability growth models

Hence, we need to measure appropriate time counts. Furthermore, a direct relation to a definition model is missing. Hence, if the reliability of our system is predicted as too low, it is not clear what we have to do to improve it. We will see a detailed example in Sect. 2.6.3.

2.1.6 Multi-Purpose Models

An ideal model in the DAP classification is a quality model that integrates all three purposes. Such a *multi-purpose model* has the advantage that we assess and predict on the same model that we use to define the quality requirements. It ensures a high consistency which might be endangered by separate definition and assessment models. One of the rare examples of a multi-purpose model is the already-mentioned *COQUAMO*.

COQUAMO was a big step forward in the 1980s to describe quality measures quantitatively and to relate measures of the product, process or organisation to measures of the product's quality using statistical models. It drew inspiration from COCOMO [23], Boehm's effort estimation model. Especially the prediction part of COQUAMO reflects this inspiration. Figure 2.3 shows the three main measures used in COQUAMO: *quality factor metrics*, *measures of quality drivers* and *quality indicators*. Each of these measures has its own purpose. The quality factor metrics measure quality factors directly. For example, the quality factor *reliability* could be measured by the *mean time to failure*. The quality factors with their quality

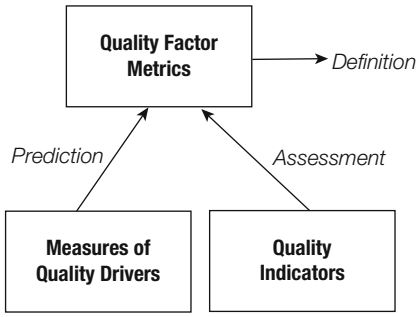

Fig. 2.3 Definition, assessment and prediction in COQUAMO

factor metrics define quality and, together with target values, can be used to specify quality requirements. The quality factors are similar as in other, hierarchical quality models: usability, testability or maintainability.

Similar to COCOMO, we then can estimate quality drivers using *measures of quality drivers* for an early prediction of the final quality of the software. These quality drivers include product attributes, such as the quality requirements; process attributes, such as the process maturity; or attributes of the personnel, such as experience. COQUAMO establishes relationships between these quality drivers and the quality factors. The idea is that a project manager can predict and adjust the drivers early in a project so that the quality requirements are achievable.

The *quality indicators* are used for assessing the quality of the product during its development to monitor and control quality. These quality indicators measure attributes of the product directly, and through established statistical relationships to quality factors, they should give an indication of the current quality. For example, a quality indicator can be the number of called modules in a module or McCabe's cyclomatic complexity [147].

A further interesting concept lies in the different measurements of quality indicators for a quality factor over the life cycle of a software. In the beginning, we should use requirements checklists, which includes standards and metrics, then design checklists, coding checklists and, finally, testing checklists. This reflects that different artefacts produced over time can contain information we can use to analyse quality.

Overall, the difficulty in COQUAMO lies in the unclear relationships of the different types of measurements. There are no functional relationships from quality indicators, such as the number of called modules, and quality factor metrics, such as the mean time to failure. Only statistical relationships can be investigated and even they are very hard to establish as many contextual factors play a role. Later, Kitchenham et al. [121] concluded from COQUAMO that "there were no software product metrics that were, in general, likely to be good predictors of final product qualities". Nevertheless, there are influences from quality indicators and quality drivers to quality factors, but a direct prediction remains difficult.

2.1.7 Critique

All the existing quality models have their strengths and weaknesses. Especially the latter have been discussed in various publications. We summarise and categorise these points of criticism. Please note that different quality models are designed for different intentions and therefore not all points are applicable to all models. However, every quality model has at least one of the following problems [49].

General

One of the main shortcomings of many existing quality models is that they do not conform to an explicit meta-model. Hence the semantics of the model elements is not precisely defined and the interpretation is left to the reader.

Quality models should act as a central repository of information regarding quality, and therefore, the different tasks of quality engineering should rely on the same quality model. Today, most quality models are not integrated into the various tasks connected to quality. For example, the specification of quality requirements and the evaluation of software quality are usually not based on the same models.

Another problem is that today quality models do not address different views on quality. In the field of software engineering, the value-based view is typically considered of high importance [201] but is largely missing in current quality models [122].

Moreover, the variety in software systems is extremely large, ranging from huge business information systems to tiny embedded controllers. These differences must be accounted for in quality models by defined means of customisation. In current quality models, this is not considered [72, 119, 156].

Definition Models

Existing quality models lack clearly defined decomposition criteria that determine how the complex concept "quality" should be decomposed. Most definition models depend on a taxonomic, hierarchical decomposition of quality factors. This decomposition does not follow defined guidelines and can be arbitrary [27, 51, 121, 122]. Hence, it is difficult to further refine commonly known quality attributes such as availability. Furthermore, in large quality models, unclear decomposition makes locating elements difficult, since developers might have to search large parts of the model to assert that an element is not already contained in it. This can lead to redundancy due to multiple additions of the same or similar elements.

The ambiguous decomposition in many quality models is also the cause of overlaps between different quality factors. Furthermore, these overlaps are often not explicitly considered. For example, considering a denial of service attack, security is strongly influenced by availability which is also a part of reliability; code

quality is an important factor for maintainability but is also seen as an indicator for security [8].

Most quality model frameworks do not provide ways for using the quality models for constructive quality assurance. For example, it is left unclear how the quality models should be communicated to project participants. A common method of communicating such information is guidelines. In practice, guidelines, which are meant to communicate the knowledge of a quality model, are not actually used. This is often related to the quality models itself; e.g. the guidelines are often not sufficiently concrete and detailed or the document structure is random and confusing. Also rationales are often not given for the rules the guidelines impose.

Assessment Models

The already-mentioned unclear decomposition of quality factors is in particular a problem for analytical quality assurance. The given quality factors are mostly too abstract to be straightforwardly checkable in a concrete software product [27, 51]. Because the existing quality models neither define checkable factors nor refinement methods to get checkable factors, they are hard to use in measurement [69, 122].

In the field of software quality, a great number of measures have been proposed, but these measures face problems that also arise from the lack of structure in quality models. One problem is that despite defining measures, the quality models fail to give a detailed account of the impact that specific measures have on software quality [122]. Due to the lack of a clear semantics, the aggregation of measure values along the hierarchical levels is problematic. Another problem is that the provided measures have no clear motivation and validation. Moreover, many existing approaches do not respect the most fundamental rules of measurement theory (see Sect. 2.2) and, hence, are prone to generate dubious results [61].

It has to be noted that measurement is vital for any quality control. Therefore the measurement of the most important quality factors is essential for a effective quality assurance processes and for a successful requirements engineering.

Prediction Models

Predictive quality models often lack an underlying definition of the concepts they are based on. Most of them rely on regression or data mining using a set of software metrics. Especially regression models tend to result in equations that are hard to interpret [62]. Also the results of data mining are not always easy to interpret for practitioners although some data mining techniques especially support this by showing decision trees.

Furthermore, prediction models tend to be strongly context dependent, also complicating their broad application in practice. Some researchers now particularly investigate local predictions [151]. It seems it is not possible to have one single set of metrics to predict defects, as found, for example, by Nagappan, Ball and

Zeller [163]. Many factors influence the common prediction goals and especially which factors are the most important ones varies strongly. Often these context conditions are not made explicit in prediction models.

Multi-Purpose Models

Although multi-purpose models provide consistency between definition, assessment and prediction, they are not yet widely established. The REQUEST project, which developed COQUAMO, took place in the 1980s. Despite that, thirty years later there is no established integrated multi-purpose quality model for software. A problem encountered in the REQUEST project was that it was difficult to find clear relationships between metrics as in other prediction models. Furthermore, I suspect that there is a large effort associated with building, calibrating and maintaining such a model. Finally, it might have also played a role that information about these old models is scarce and not easily available to software engineers.

2.1.8 Usage Scenarios

In summary and to understand how we want to use software quality models, we will briefly discuss the main usage scenarios [49]:

Definition models are used in various phases of a software development process. During requirements engineering, they define quality factors and requirements for planned software systems [122, 204] and, thus, constitute a method to agree with the customer what quality means [122]. During implementation, quality models serve as basis of modelling and coding standards or guidelines [54]. They provide direct recommendations on system implementation and, thus, constitute constructive approaches to achieve high software quality. Furthermore, quality defects that are found during quality assurance can be classified using the quality model [54]. Apart from their use during software development, quality definition models can be used to communicate software quality knowledge during developer training or student education.

Assessment models often naturally extend quality definition model usage scenarios to control compliance. During requirements engineering, assessment models can be used to objectively specify and control stated quality requirements [122]. During implementation, the quality model is the basis for all quality measurements, i.e. for measuring the product, activities and the environment [54, 197, 202]. This includes the derivation of guidelines for manual reviews [51] and the systematic development and usage of static analysis tools [54, 174]. During quality audits, assessment models serve as a basis of the performed audit procedure. Thereby, internal measures that might influence external properties are monitored and controlled [122]. Apart from their use during software development, assessment models furthermore constitute the touchstone for quality certifications.

Prediction models are used during project management. More specifically, such models are used for release planning and in order to provide answers to the classical "when to stop testing" problem [157]. *Multi-purpose* models, finally, combine all the above usage scenarios.

2.1.9 Summary

An impressive development of quality models has taken place over the last decades. These efforts have resulted in many achievements in research and practice. As an example, consider the field of software reliability engineering that performed a wide as well as deep investigation of reliability growth models. In some contexts these models are applied successfully in practice. The developments in quality definition models led to the standardisation in ISO/IEC 25010 that defines well-known quality factors and will serve as the basis for many quality management approaches. It even integrates with a quality evaluation method in ISO/IEC 25040, so that the international standard describes a multi-purpose quality model to some degree.

The whole field of software quality models, however, remains diverse and fuzzy. There are large differences between many models that are called "quality models". Moreover, despite the achievements made, there are still open problems, especially in the adoption in practice. Because of this, current quality models are subject to a variety of points of criticism that will keep further generations of software quality researchers busy.

2.2 Software Measures

The area of software metrics has been under research from the early days of software engineering. It describes how to measure this abstract thing "software", hopefully in a way that helps the software engineers to develop software better and faster. As we talk about measurement, we adopted the term "measure" instead of "metric" in most parts of the book, but we will use it interchangeably. In this section, we discuss the basics of software metrics with interesting properties and scales and then focus on aggregation because that is very important for quality evaluation.

2.2.1 Basics

Measurement is the mapping from the empirical world to the formal world. In the empirical world, there are entities (things) that have certain attributes. These attributes can be expressed with measures from the formal world. Measurement

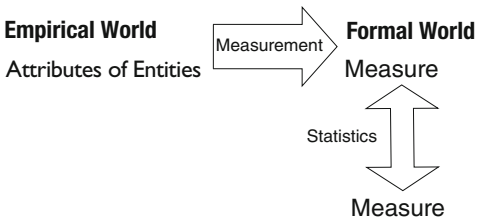

Fig. 2.4 The general concepts of measurement and statistics

theory is therefore responsible for arguing about the relationship between reality and measures. For example, the table in my office is certainly an entity of reality and has the attribute *height*. Measurement maps this real attribute of the table to the formal world by stating that the height is 71 cm. Transferred to software this means that the entity *source code* has the attribute *length* which we can measure in lines of code (LOC). This relationship is depicted in Fig. 2.4.

Why would we be interested in this transformation into the formal world? Because we now can analyse the measures and, in particular, the relationships between different measures. Describing the relationships between those different measures is the task of statistical theory. For example, we can investigate and describe the relationship between the height of tables and their users. Grown-up "table users" will have, on average, higher tables than children in their rooms or in the class room. Similarly, software with more LOC has probably taken more effort (e.g. measured in person month) than software with less LOC.

We will come across some statistical theory below in the context of aggregation. Also for measurement, there is a well-proven measurement theory that helps us in avoiding mistakes in measuring and interpreting measurements. We will in particular discuss scales and important properties of measures. Scales are probably the most important part of measurement theory, because they can help us to avoid misinterpretations.

Scales

In principle, there is an unlimited number of scales possible for software engineering measures. It usually suffices, however, to understand five basic scales to be able to interpret most measures:

1. Data that only gives names to entities has a *nominal scale*. Examples are defect types.
2. If we can put the data in a specific order, it has an *ordinal scale*. Examples are ratings (high, medium, low).
3. If the interval between the data points in that order is not arbitrary, the scale is *interval*. An example is temperature in Celsius.
4. If there is a real 0 in the scale, it is a *ratio* scale. An example are LOC.

5. If the mapping from the empirical world is unique, i.e. there is no alternative transformation, it is an *absolute* scale. An example is the ASCII characters in a file.

To understand in which of these scales your measures are is necessary to do the right interpretation. For example, temperature measured in Celsius is in an interval scale. The intervals between degrees Celsius are not arbitrary but clearly defined. Nevertheless, it does not make sense to say "It is twice as hot as it was yesterday!" if it has 30 degrees today and it had 15 degrees yesterday. Another perfectly valid unit for temperature is Fahrenheit in which, for the same example, it would be 59 Fahrenheit yesterday and 86 Fahrenheit today. That is not "twice as hot". The reason is that the 0 in these scales was artificially chosen. In contrast, LOC has a real 0 when there are no lines of code. Therefore, it is permissible to say "This software is twice as large as the other!"

Therefore, the scales define what is permissible to do with the data. We will use the scales later to discuss what aggregations are possible below in detail. For example, for measures with a nominal scale, we cannot do much statistical analysis but count the numbers of the same values or find the most often occurring value (called the *mode*). For ordinal data, I have an order and, therefore, I can find an average value which is the value "in the middle" (called the *median*). From an interval scale on, we can calculate a more common average, the *mean*, summing all the values and dividing by the number of values. With ratio and absolute scales, we can use any mathematical and statistical techniques available.

Properties of Measures

Apart from the scale of a measure, we should consider further properties to understand the usefulness and trustworthiness of a measure. Very important desired properties of measures are reliability and validity. *Reliability* means in this context that the measure gives (almost) the same result every time it is measured. *Validity* means that its value corresponds correctly to the attribute of the empirical entity. Different possibilities for the reliability and validity of a measure are illustrated in Fig. 2.5. A measure is neither reliable nor valid if it produces a different value every time it is measured and none of these describes the attribute of the empirical entity well (left in Fig. 2.5). A measure can be reliable by producing similar results in every measurement but not be valid because the reliable measurements do not correspond to the empirical reality (middle). Finally, we want a measure which is reliable and valid (right). These two properties are expected to be achieved with any measure we define. Validity is often difficult to show and reliability can be tricky for manually collected measures which are often subjective.

In addition, there are further properties of measures which are also desired but not always possible to be achieved. As already discussed above, the reliability of a measurement can be problematic for subjective measures. Therefore, we aim for *objectivity* in measures meaning that there is no subjective influence in

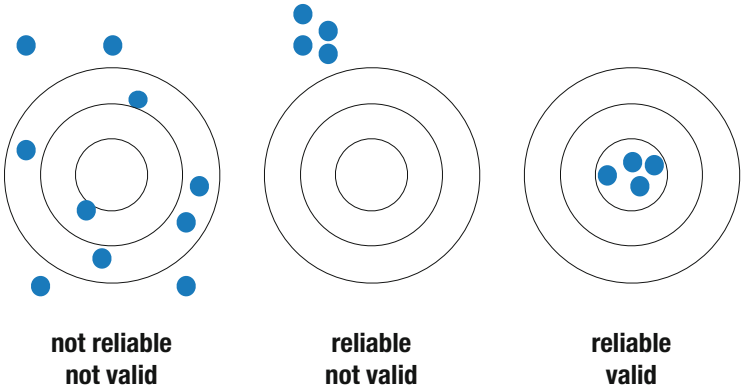

Fig. 2.5 Reliability and validity of measures

measurement. Next, we want to be able to use the measure in the formal world and compare it to other measures (*comparability*). This requires a suitable scale for the measure (*standardisation*). Moreover, we can measure countless things in a software development project, but the *usefulness* of those measure should be ensured in the sense that they fulfil practical needs. Finally, *economy* is also a desired property for measures. It is helpful to be able to collect them with low cost. Then, we can collect and analyse often and early. We have observed, however, that in many cases the more useful the measures are, the less economic they are.

2.2.2 Aggregation

A plethora of measures for software has been proposed that use numerous properties and aim at various purposes. Several of these measures are independent of the granularity of the measured software entity. For example, lines of code can be measured for a whole system or for subcomponents only. Many of these measure, however, are "local", i.e. measure properties that are only useful for a specific part of the software. For example, the depth in the inheritance tree [34] can only be calculated for a single class in an object-oriented system. Aggregating these measures is not straightforward but necessary for quality evaluation. We will introduce here how we can apply aggregation operators to software measures and provide a collection of possible operators for your aggregation tasks. These operators are categorised according to the suitability for different purposes. We also describe the implementation of aggregation in a dashboard toolkit and use several of the operators in an example.

To have a model or goal for a measure is common in research to ensure its usefulness [10, 20, 57]. Building these models often includes the aggregation of measures to define higher-level measures. For example, it can be interesting to aggregate

measures defined for classes to the system level to be able to compare different systems. As discussed in [86], the measurement system "Emerald aggregates the metrics [...] to a level specified by the user. The aggregation levels include source file, module, test configuration, development group, release, and major subsystem".

This point is also emphasised by Blin and Tsoukiàs [20]: "...the definition of an evaluation model implies a number of choices including the aggregation procedures to be used. Such choices are not neutral and have to be rigorously justified. Otherwise, results are very difficult to understand and to exploit, when they are not totally surprising". Hence, aggregation is an important part of any measurement system and a reoccurring task in any measurement method.

However, as Blin and Tsoukiàs also state in [20]: "It is surprising how often the choice of the aggregation operator is done without any critical consideration about its properties". For example, in the IEEE standard 1061 [87], only weighted sums are mentioned for aggregation. Yet, the choice of the right aggregation operator has a strong influence on the results and hence needs to be justified carefully.

Aggregation Purposes

A justification of the suitability of a specific aggregation operator is only possible on the basis of the purpose of the aggregation. It is completely different if the current state of a system should be assessed or whether its hot spots should be identified. A complete list of aims of aggregation (and thereby measurement) is difficult to work out, but we identified five different aims that cover most cases. This categorisation is partly similar to the categorisation of purposes of quality models in Sect. 2.1. Note that they also build on each other and that more sophisticated analyses presuppose the simpler ones.

Assessment

The first and most common purpose of collecting data about a system is to assess its current state. This starts with simple measures such as size or age but comes quickly in extremely complex areas such as the reliability or the maintainability of the system. Important in assessment is that we do not take the future into account, at least not directly. Nevertheless, when determining a size trend in the assessment, it might have implications for the future as well. We are not interested in how the size grew over the time the system has existed, however. This is what in software reliability engineering is often called *estimation* [139].

There are three different modes in which assessments of software can be done. First, a singular, unique "health check" is a possibility in which the project or quality manager wants an overview of the system and of a specific set of its properties. Second, in process models for software development, there are often defined *quality gates* (Sect. 4.1) where the quality of certain artefacts needs to be checked. Third, it is possible to implement suitable assessments as part of the daily or weekly build

of the software. This way, problems are detected early and can be confronted when they are still relatively cheap. For example, in our continuous quality control, several measures are routinely measured to check the current quality.

Prediction

When assessments are in place and possible for certain software properties, the next desired step is to predict the future development of these properties. This is what most software reliability models are concerned with [139]. Here, we use information about the current state of the system, possibly enriched with historical data about the development of measures, to determine a probable future state of the system. Hence, usually there are probabilistic models involved in this prediction process. In prediction, we might need to aggregate the predicted results from subsystems to systems, for example.

As mentioned, reliability growth models are well-known examples that use historical defect data to predict the reliability in the field [136, 159, 207]. Another example in the context of maintenance is models that predict the future maintenance efforts for a software system [81, 221].

Hot Spot Identification

Often, it is interesting not only to measure a certain property now or predict its development but also to detect specific areas of interest, denoted as "hot spots" here. This is especially important after the assessment and/or prediction to derive countermeasures against the detected problems. For a specific instruction for improvement, we need to know where to start with the improvement. Usually, we want to know where is the biggest problem and hence the largest return on investment when there is improvement.

For example, there are various methods for identifying fault-prone components [162, 208] using different approaches based on software measures. Also here the problem of aggregation can easily be seen: Is a component fault prone when its subcomponents are fault prone?

Comparison

Another important purpose is to compare different components or systems based on certain (aggregated) measures. In principle, comparisons are simply assessments of more than one software. The difficulty lies in comparability of the measures used for comparison, however. Not all measures are independently useful without having its context. Hence, normalisations need to take place for this. A common example is normalisations by size as in the defect density that measures the average defects per LOC.

Comparisons are useful for project managers to analyse the performance of the project. It can be compared with other similar systems or the state-of-the-art, if there are, published values. Jones [109] has industry averages for various measures. Another example is the classical *make or buy* decision about complete systems or software components. If there are measurements possible for the different available components, they can be used to aid that decision.

Trend Analysis

Finally, there are cases in which we want not only a measure at a specific point in time but also its changes over time. This is somehow related to predictions, as we discussed with reliability growth models. They also use the change over time of occurred failures to predict the future. It is also a basic tool in *statistical process control* SPC [65]. It is analysed whether the analysed data varies in a "normal" span or if there are any unusual events. For example, the growth rate of defects or code size can be useful for project management. Another example is the number of anomalies reported by a bug pattern tool or the number of clones found by clone detection. There might not be enough resources to fix all these issues, but at least it should be made sure that the number is not increasing anymore. Again, there are various aggregation issues about determining the correct measure for a component from its subcomponents, for example.

Aggregation Theory

Aggregation is not only a topic for software measures but in any area that needs to combine large data into smaller, more comprehensible or storable chunks. We describe the general theory of aggregation aggregators first and then discuss the state in software engineering.

General Theory

There is a large base of literature on aggregation in the area of soft computing [18, 31] where it is also called *information fusion*. They use aggregation operators in the construction and use of knowledge-based systems. "Aggregation functions, which are often called aggregation operators, are used to combine several inputs into a single representative value, which can be subsequently used for various purposes, such as ranking alternatives or combining logical rules. The range of available aggregation functions is very extensive, from classical means to fuzzy integrals, from triangular norms and conorms to purely data driven constructions" [17]. We mainly build on the variety of work in that area in the following to explain what is now considered the basics of aggregation. We mainly use the classification and notation from [53] and [18].

Informally, aggregation is the problem of combining n-tuples of elements belonging to a given set into a single element (often of the same set). In mathematical aggregation, this set can be, for example, the real numbers. Then an aggregation operator A is a function that assigns an y to any n-tuple $(x_1, x_2, \ldots, x_n)$:

$$A(x_1, x_2, \ldots, x_n) = y \tag{2.1}$$

From there on, the literature defines additional properties that are requirements for a function to be called an *aggregation operator*. These properties are not all compatible, however. Yet, there seem to exist some undisputed properties that must be satisfied. For simplification, the sets that aggregation operators are based on are usually defined as $[0, 1]$, i.e. the real numbers between 0 and 1. Other sets can be used, and by normalisation to this set, it can be shown that the function is an aggregation operator. Additionally, the following must hold:

$$A(x) = x \text{ identity when unary} \tag{2.2}$$

$$A(0, \ldots, 0) = 0 \wedge A(1, \ldots, 1) = 1 \text{ boundary conditions} \tag{2.3}$$

$$\forall x_i, y_i : x_i \leq y_i \Rightarrow$$
$$A(x_1, \ldots, x_n) \leq A(y_1, \ldots, y_n) \text{ monotonicity} \tag{2.4}$$

The first condition only is relevant for unary aggregation operators, i.e. the tuple that needs to be aggregated only has a single element. Then we expect the result of the aggregation to be that element. For example, the aggregation of the size in LOC of a single Java class should be the original size in LOC of that class.

The boundary conditions cover the extreme cases of the aggregation operator. For the minimum as input there must be the minimum as output and vice versa. For example, if we aggregate a set of ten modules, all with 0 LOC, we expect that the aggregation result is also 0 LOC.

Finally, we expect that an aggregation operator is monotone. If all values stayed the same or increased, we want the aggregation result also to increase or at least stay the same. This is interesting for trend analyses. If we aggregate again the size of modules to the system level and over time one of the modules increased in size (and none decreased), we want also the aggregation of the size to increase.

Apart from these three conditions, there is a variety of further properties that an aggregation operator can have. We only introduce three more that are relevant for aggregation operators of software measures.

The first condition that introduces a very basic classification of aggregation operators is *associativity*. An operator is associative if the results stay the same no matter in what packaging the results are computed. This has interesting effects on the implementation of the operator as associative operators are far easier to compute. Formally, for an associative aggregation operator A_a the following holds:

$$A_a(x_1, x_2, x_3) = A_a(A_a(x_1, x_2), x_3) = A_a(x_1, A_a(x_2, x_3)) \tag{2.5}$$

The next interesting property is *symmetry*. This is also known as *commutativity* or *anonymity*. If an aggregation operator is symmetrical, the order of the input arguments has no influence on the results. For every permutation σ of $1, 2, \ldots, n$ the operator A_s must satisfy

$$A_s(x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(n)} = A_s(x_1, x_2, \ldots, x_n) \tag{2.6}$$

The last property we look at because it holds for some of the operators relevant for software measures is *idempotence*. It is also known as *unanimity* or *agreement*. Idempotence means that if the input consists of only equal values, it is expected that the result is also this value.

$$A_i(x, x, \ldots, x) = x \tag{2.7}$$

An example for a an idempotent software measure is clone coverage (Sect. 4.4.2) which describes the probability that a randomly selected line of code is copied in the system. We can calculate the clone coverage for each individual module, and if we aggregate modules with the same clone coverage, we expect the aggregation result also to be this exact same clone coverage.

Software Engineering

Only few contributions to the theory of aggregation operators in software measurement have been made. The main basis we can build on is the assignment of specific aggregation operators (especially for the central tendency) to scale types. Representational theory has been suggested as basis for a classification of scale types for software measures [61]. The scales are classified into *nominal*, *ordinal*, *interval*, *ratio* and *absolute*. Nominal scales assign only symbolic values, for example, *red* and *green*. They are introduced in detail above.

This classification provides a first justification for which aggregation operators can be used for which classes of scales. For example, consider the measures of central tendency such as median or mean. To calculate the mean value of a nominal measure does not make any sense. For example, what is the mean of the names of the authors of modules in a software system? This is only part of the possible statistics that we can use as aggregation operators. Nevertheless, we will use this classification as part of the discussion on the suitability of aggregation aggregators.

Aggregation Operators

Grouping

A very high-level aggregation is to define a set of groups, probably with a name each, and assign the inputs to the groups. This allows a very quick comprehension and easy communication about the results. Problematic is that the information loss is rather large.

Rescaling

An often used technique to be able to comprehend the large amount of information provided by various measures is to change the scale type by grouping the individual values. This is usually done from higher scales, such as ratio scales, to ordinal or nominal scales. For example, we could define a threshold value for test coverage. Above the threshold, the group is *green*; below it, it is *red*. This is useful for all purposes apart from trend analysis where it can be applied only in a few cases. It is not idempotent in general and it depends on the specifics of the rescaling whether it is symmetrical.

Cluster Analysis

Another, more sophisticated way, to find regularities in the input is cluster analysis. It is, in some sense, a refinement of the rescaling described above by finding the groups using clustering algorithms. The *K-means* [140] algorithm is a common example of such algorithms. It works with the idea that the input are points scattered over a plain and there is a distance measure that can express the space between the points. The algorithms then work out which points should fall into the same cluster. This aggregator is not associative and not idempotent.

Central Tendency

The central tendency describes what colloquially is called the average. There are several aggregation operators that we can use for determining this average of an input. They depend on the scale type of the measures they are aggregating. All of them are not associative but idempotent.

The *mode* is the only way for analysing the central tendency for measures in a nominal scale. Intuitively, it gives the value that occurs most often in the input. Hence, for inputs with more than one maximum, the mode is not uniquely defined. If the result is then defined by the sequence of inputs, the mode is not symmetrical. The mode is useful for assessing the current state of a system and for comparisons of measures in a nominal scale. For $n_1, \ldots, n_k$ being the frequencies of the input values, the mode M_m is defined as

$$M_m(x_1, \ldots, x_k) = x_j \Leftrightarrow n_j = \max(n_1, \ldots, n_k). \tag{2.8}$$

The *median* is the central tendency for measures in an ordinal scale. An ordinal scale allows to enforce an order on the values and hence a value that is in the middle can be found. The median ensures that at most 50 % of the values are smaller and at most 50 % are greater or equal. The median is useful for assessing the current state and comparisons. The median $M_{0.5}$ is defined as

$$M_{0.5}(x_1, \ldots, x_k) = \begin{cases} x_{((n+1)/2)} & \text{if } n \text{ is odd} \\ \frac{1}{2}(x_{(n/2)} + x_{(n/2+1)}) & \text{otherwise} \end{cases} \tag{2.9}$$

For measures in interval, ratio or absolute scale, the *mean* is defined. There are mainly three instances of means: arithmetic, geometric and harmonic mean. The arithmetic mean is what usually is considered as average. It can be used for assessing the current state, predictions and comparisons. The arithmetic mean M_a is defined as follows:

$$M_a(x_1, \ldots, x_n) = \frac{1}{n} \sum_{i=1}^{n} x_i \tag{2.10}$$

We can use the geometric mean for trend analysis. It is necessary when measures are relative to another measure. For example, the growth rates of the size of several releases could be aggregated using the geometric mean. The geometric mean M_g is defined as

$$M_g(x_1, \ldots, x_n) = \sqrt[n]{\prod_{i=1}^{n} x_i}. \tag{2.11}$$

As it uses the product, it actually has the absorbent element 0 [53]. Finally, the harmonic mean needs to be used when different sources are combined and hence weights need to harmonise the values. An example would be when, to analyse the reliability of a system, the fault densities of the components are weighted based on their average usage. Given the weights w_i for all the inputs x_i, the harmonic mean M_h is given by

$$M_h(x_1, \ldots, x_n) = \frac{w_1 + \ldots + w_n}{\frac{w_1}{x_1} + \ldots \frac{w_n}{x_n}}. \tag{2.12}$$

Dispersion

In contrast to the central tendency, the dispersion gives an impression about how scattered the inputs are over their base set. Hence, we look at extreme values and their deviation from the central tendency.

The *variation ratio* is given by the proportion of cases which are not the mode. This is the only way for nominal measures to have a measure of dispersion. It indicates whether the data is in balance. This is useful for hot spot identification. The variation ratio V is defined again using $(n_1, \ldots, n_k)$ as the frequencies of $(x_1, \ldots, x_k)$ by

$$V(x_1, \ldots, x_k) = 1 - \frac{\max(n_1, \ldots, n_k)}{k}. \tag{2.13}$$

Very useful operators for various analysis situations are the *maximum and minimum* of a set of measures. They can be used with measures of any scale apart from nominal. They are useful for identifying hot spots and for comparisons. They both are associative and symmetrical. The maximum *max* and the minimum *min* are defined as follows:

$$\forall x_i . y \geq x_i \Rightarrow \max(x_1, \ldots, x_n) = y \tag{2.14}$$

$$\forall x_i . y \leq x_i \Rightarrow \min(x_1, \ldots, x_n) = y \tag{2.15}$$

The *range* is the standard tool for analysing the dispersion. Having defined the maximum and the minimum above, it is easy to compute. It is given by the highest value minus the lowest value in the input. It can be useful for assessing the current state and comparisons. This operator is neither idempotent nor associative. The range R is simply defined as

$$R(x_1, \ldots, x_n) = \max(x_1, \ldots, x_n) - \min(x_1, \ldots, x_n) \tag{2.16}$$

The *median absolute deviation* (MAD) is useful for ordinal metrics. It is calculated as the average deviation of all values from the median. This again can be used for current state analyses and comparisons when the median is the most useful measure for the central tendency. The median absolute deviation is not associative but symmetrical. It is defined as

$$D(x_1, \ldots, x_n) = \frac{1}{n} \sum_{i=1}^{n} |x_i - M_{0.5}(x_1, \ldots, x_n)|. \tag{2.17}$$

For other scales, the most common dispersion measures are the *variance* and the *standard deviation*. It is always the best choice for analysing the dispersion when the mean is the best aggregator for the central tendency. The standard deviation has the advantage over the variance that it has the same unit as the measure to be aggregated. Hence, it is easier to interpret. We use it again for analysing the current state and for comparisons. It is not associative but symmetrical. The variance S^2 and the standard deviation S are defined as follows:

$$S^2(x_1, \ldots, x_n) = \frac{1}{n} \sum_{i_1}^{n} (x_i - M_a(x_1, \ldots, x_n))^2 \tag{2.18}$$

$$S(x_1, \ldots, x_n) = \sqrt{S^2(x_1, \ldots, x_n)} \tag{2.19}$$

Concentration

A step further than central tendency and dispersion are concentration aggregators. They measure how strongly the values are clustered over the whole span of values.

For concentration measures to be available, the measures need to be at least in interval scale. The basis for these concentration measures is often the *Lorenz curve*. It describes the perfect equality of a distribution. The Lorenz curve itself is built from the ordered inputs but alone can only give a graphical hint of the concentration.

The *Gini coefficient* is a concentration measure that is defined by the deviation of the actual distribution from the perfect Lorenz curve. Its normalised version measures the concentration in a scale between 0 (no concentration) and 1 (complete concentration). This can be used for a more sophisticated analysis of hot spots. If the operator assumes that the input is preordered, the Gini coefficient is not symmetrical. It is also not associative. The normalised Gini aggregation operator G_n where (i) indicates the ith element in the ordered input is defined as

$$G_n = \frac{2\sum_{i=1}^{n} i\,x_{(i)} - (n+1)\sum_{i=1}^{n} x_{(i)}}{n\sum_{i=1}^{n} x_{(i)}} \tag{2.20}$$

Ratios and Indices

A further way to quantitatively describe aspects that can be used as aggregation operators are ratios and indices. They are binary operators and therefore relate two different measures. It is important that the results of such operators are usually without a dimension or unit. This is because they only describe a relation. Because they are all defined as ratios, they are not associative but usually they are symmetrical. The calculation is straightforward; therefore, we refrain from giving explicit equations for each aggregation operator. The differences in this operators do not lie in the different calculation but are conceptual differences.

Fraction measures relate subsets to their superset. An example would be *comment lines/total lines of code*. It is suitable for assessing the current state or to make comparisons.

Relation measures are measures of two different metrics that are not in a subset–superset relation. An example is *deleted lines/remaining lines*. This is again interesting for comparisons or for trend analyses.

Indices describe the relationship between the result of a metric and a base set measured at different points in time. This fits to trend analyses. For example, the *produced lines of code* each month in relation to the *lines produced on average* in the company are an index.

Tool Support

The aggregation operators presented above can easily be implemented and included in quality evaluation tools. We did that prototypically for the tool ConQAT[5] [48]

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

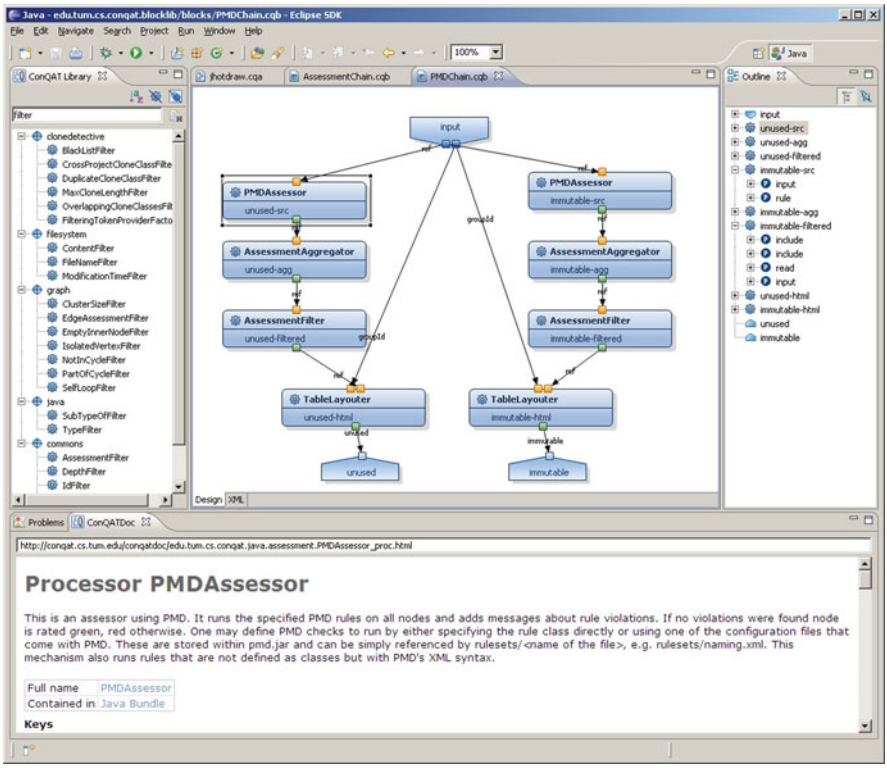

Fig. 2.6 The graphical editor for ConQAT with an example configuration

which we also use as dashboard for continuous quality control. Furthermore, it implements the Quamoco quality evaluation approach (Sect. 4.2). It makes use of a pipes and filters architecture in which complex analyses can be built by connecting simple so-called *processors*. These processors read the artefacts of interest, perform analyses on them and write the resulting outputs. There is also a graphical editor in which the nets of processors can be modelled and run. An example is shown in Fig. 2.6.

ConQAT already provides various processors for different languages and artefacts, such as Java and C# source code, Matlab Simulink models or Bugzilla defect databases. It has been used in several industrial environments for a variety of analyses [48]. Aggregation is an important part of combining these processors.

The implementation of the aggregation operators can be done straightforwardly as a ConQAT processor each. The list of values to be aggregated is the input in the aggregator that calculates the result of the operator and writes it in the output. More interesting is the case when we operate on a hierarchy such as Java classes in packages. Then the aggregation can be done on each inner node in the hierarchy. Here the distinction between associative and non-associative aggregation operators becomes important. For associative operators, we only need to consider the direct

children for each inner node. Non-associative operators require a complete traversal of the subtree.

It is not always the best way to implement processors for aggregators in the strict theoretical sense as described above. In theory, an aggregation operator always maps a list of values to a *single* value. Because of computational savings and convenience of use, however, it makes often sense to combine several aggregation operators in one processor. For example, the variation ratio is easily computed together with the mode. Actually, a separate implementation would require to identify the mode again. Also note that all the ratios and indices can be implemented in one processor as their differences are purely conceptual.

Example

To demonstrate the use of the different aggregation operators and their respective advantages, we describe an example application to the open source project *Apache Tomcat*, a Java servlet container. The hypothetical context is a comprehensive quality evaluation of this software system. We briefly describe Tomcat, the investigated evaluation scenarios and the achieved results with the aggregation operators.

Apache Tomcat

Tomcat[6] is a project by the Apache Software Foundation that is also responsible for the most successful web server. Tomcat is also aiming at the web but is a servlet container, i.e. able to run servlets and Java Server Pages (JSP). Several of the best developers in that area contribute to Tomcat as it is also used in the reference implementation of the corresponding specifications.

The version 6.0.16 that we used in this example conforms to the servlet specification 2.5 and the JSP specification 2.1. It contains various functionalities needed in a web and servlet context: network management, monitoring, logging and balancing.

Analysis Scenarios

We perform a quality evaluation of Tomcat using the dashboard toolkit ConQAT including the implementation of the aggregation operators as described above. For the analysis, we need a set of scenarios that form the quality issues that are currently of interest. These span quite different aspects of the system and are aimed at showing the various possibilities of the aggregation operators. We concentrate on only three scenarios that illustrate the use of the aggregators:

[6] http://tomcat.apache.org/.

First, we analyse the *average class size*. Although a problematic measure in many aspects, *lines of code* (LOC) is reasonable to get an impression of the size of a system and of its components. To aid code reading and comprehension, it is advantageous not to have too large classes. Therefore, we want an analysis that shows us the average size of classes overall and for each package. This is purely for assessment as no direct actions can be derived from the results.

Second, we analyse the activity of the different developers by assessing who is the *most frequent author*. This can be interesting just for identifying and gratifying the most productive author. Although this should be done with care as only the authorship of classes is not a meaningful measure for productivity. If clear responsibilities for specific packages exist, however, these author aggregations can also be used to check whether the responsible persons actually work the most on their packages.

Third, one measure that is considered important in object-oriented development is the *coupling between classes* [34]. We are interested in how strongly the classes in the software are interconnected. For this, we want to know how many classes each class depends on in relation to the size of the class. The latter requirement comes from the fact that the dependence on other class can clearly be influenced by its size: a larger class potentially relies on more classes.

Results

The corresponding ConQAT configurations for the scenarios were developed by combining existing ConQAT processors with the aggregation operators as described above. Actually, we developed one configuration that wrote the results for the three scenarios in one HTML output.

Size analyses are a practical way to get a feeling for a system. The *average class size* is a suitable measure for that purpose when analysing Java software. For the analysis, we use the mean operator and the range operator. This way, we see not only the averages for the classes but also some information about the dispersion of the values. For illustration, the output of ConQAT is shown in Fig. 2.7. The package `org.apache.tomcat`, for example, has a mean class size of 174.724 LOC with the large range of 2,847 LOC. In contrast, the package `org.apache.catalina.tribes.transport.bio` has the higher mean of 181.111 LOC but only a range of 224 LOC. Hence, the sizes of the classes in the latter package are much more uniform.

We do not have a list of responsibilities for the packages of Tomcat. Hence, we only analyse which are the *most frequently found authors* in which packages. First of of all, it becomes obvious with this analysis whether the *author* tag from JavaDoc is consistently used in a project. In Tomcat it is not used in all classes but in many. As we only look at names, which are of a nominal scale, the mode operator has to be used. The application of the mode shows that actually the most frequent author is the empty one. More interesting are specific subpackages: the package `org.apache.catalina.connector` is dominated by Remy

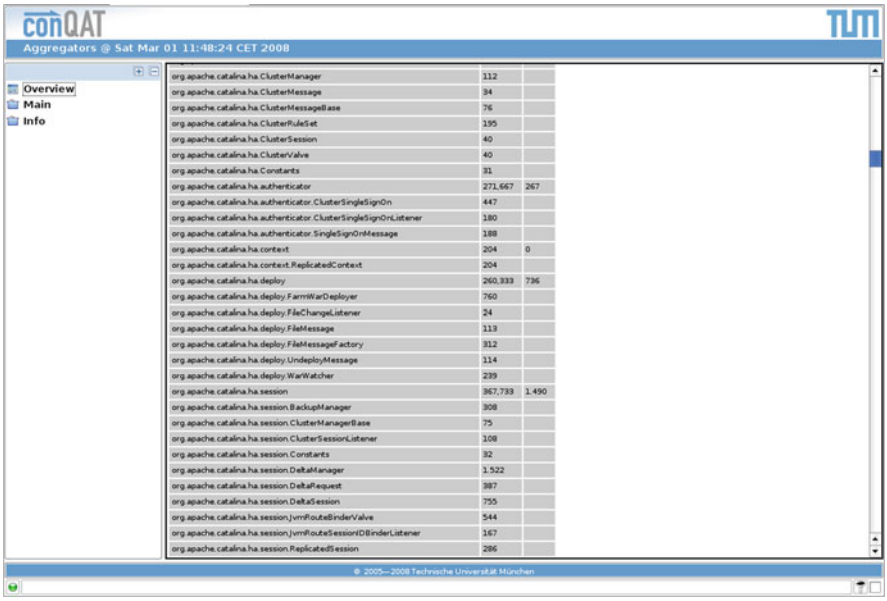

Fig. 2.7 The output of ConQAT for the average class sizes

Maucherat or `org.apache.catalina.valves` by Craig R. McClanahan. To add more information to these statements, we also used the variation ratio operator that determines the fraction of all others that are not the mode. For the complete project, we learn that although the empty author is the most frequent one, 83.3 % of the classes have named authors. In the connector package Remy Maucherat is not the author of 76.5 % of the classes, and in the valves package Craig R. McClanahan is not the author of 60 % of the classes.

We restrict the *coupling of classes* analysis to the fan-in of each class. This measures the number of classes that depend on the current class. ConQAT already provides the processor *ClassFanInCounter* for that purpose. Running that processor on the Java code of Tomcat provides us with a corresponding list. Various aggregations of these values are possible. The maximum aggregator gives the class with the highest fan-in. For Tomcat, as probably in many systems, this is the logging class `Log` with 230 classes that depend on it. Also the mean or variation is an interesting aggregation to understand the distribution of the fan-ins. Finally, we can define a relation aggregator that measures the number of fan-ins per size in LOC. This gives an indication whether larger classes are more used than smaller classes as they might provide more functionality. However, this is not the case with Tomcat.

Discussion

Our first observation is that the set of described aggregation operators is useful and sufficient in the encountered situations. Many questions can be answered with the

most basic operators for central tendency and dispersion. Most of the time several of those basic operators are used in combination to get a better overview of the software characteristic under analysis.

Moreover, it is important to point out that the three basic properties of aggregation operators work well in categorising the operators. Several operators such as the minimum or the maximum are associative; others as the mean or the median are not. This is especially useful for implementation as the associativity allows a simpler and quicker implementation. In general, the associative and the non-associative operators were implemented with two different base classes. The idempotence holds for most of the operators but usually not for rescaling. Finally, the symmetry is a property of all the operators we discussed above which is also important for their implementation and use. The user of the operators never has to worry about in what sequence the input currently is.

Summary

We gave in this section a comprehensive but certainly not complete overview of software measures and measurement as well as a collection of useful aggregation operators for use in a variety of measurement systems. We judged the suitability of each operator based on a set of aggregation purposes because we see the purpose of the aggregation as the main driver of the operator choice. We also discussed further properties of operators such as associativity that are important for implementation and use of the operators. Moreover, the scale type of the measures is a further constraint that limits the available operators for a specific measure.

2.3 ISO/IEC 25010

After several years of development, a working group of the International Organization for Standardization released in 2011 a reworked software product quality model standard: ISO/IEC 25010 [97]. It is still strongly influenced by its predecessor ISO 9126 [107] but restructures and adds several parts of the quality models. This standard is likely to become the most well-known type of software quality models, and it will have an effect on existing quality models in practice. We therefore look into it in more detail.

2.3.1 Concepts and Meta-Model

ISO/IEC 25010 grew historically from the initial hierarchical models from Boehm and McCall which were already the basis of the old ISO/IEC 9126 (cf. Sect. 2.1). The main modelling concept is hence to provide a taxonomy that breaks the complex

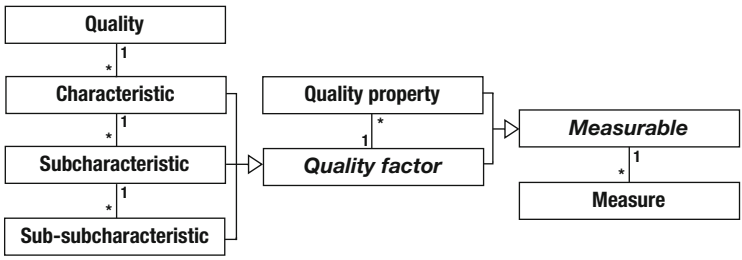

Fig. 2.8 Implicit meta-model of ISO/IEC 25010

concept of software product quality into smaller, hopefully more manageable parts. The idea is that the decomposition reaches a level on which we can measure the parts and use that to evaluate the software product's quality.

A UML class diagram of the meta-model of the ISO/IEC 25010 quality models is shown in Fig. 2.8. It shows the hierarchical structure that divides quality into *characteristics*, which can consist of *subcharacteristics* and, in turn, of *sub-subcharacteristics*. We call all of these *quality factors* and they might be measurable by a *measure*. If a direct measurement is not possible, we use measurable *quality properties* that cover the quality factor. Please note that the terms *quality factor* and *measurable* are not part of the standard, but we introduced them here for the sake of a well-structured meta-model.

Using this hierarchical structure, the standard then assumes that we can build a quality model. ISO/IEC 25010 contains a quality in use model and a product quality model which we will cover in the next sections. In addition, there is a compliant data quality model in ISO/IEC 25012 [98]. These models shall help the software engineers as a kind of checklist so that they consider all relevant parts of software quality in their products. Nevertheless, the standard emphasises that not all characteristics are relevant in any kind of software. It gives no help how to customise the quality model, however.

The description of the quality model structure explicitly mentions the connection to measures. Hence, it is considered important to be able to quantify quality and its characteristics. The details of this is not part of the quality model, however. The standard ISO/IEC 25040 [102] adds quality evaluation to the models. We will discuss quality evaluation in more detail in Sect. 4.2.

2.3.2 Product Quality Model

We have used this quality model already in the introduction to discuss the various areas of software quality (Sect. 1.3.3). Furthermore, the product quality model fits best to traditional software product quality models that we discussed in Sect. 2.1. It describes the most common "-ilities" for software and several subcharacteristics for each. There is an overview in Fig. 2.9. The idea is that each of the characteristics

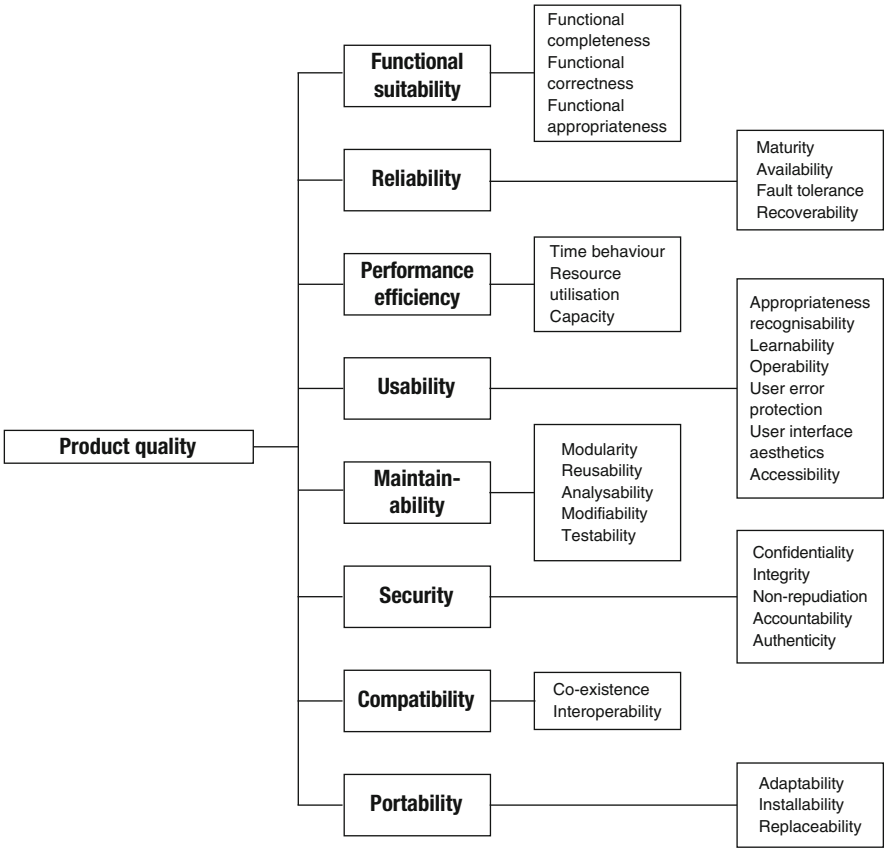

Fig. 2.9 Product quality model of ISO/IEC 25010 [97]

is something we can analyse directly at the software product. In addition, the standard claims that they should also be valid for system quality.

The intention of the standard is that this list of eight characteristics comprehensively describes the quality of a software product. *Functional suitability* means that the product fits to the functional needs and requirements of the user and customer. *Performance efficiency* describes how well the product responds to user requests and how efficient it is in its execution. Today's software products rarely operate in an isolated environment and therefore *compatibility* defines the quality that a product does not disturb or can even work together with other products. The characteristic *usability* subsumes the aspects of how easy the system can be used. This includes how fast using it can be learned but also if the interface is attractive and if challenged persons are able to use it.

Many people mean *reliability* when they talk about quality. Reliability problems are – together with performance efficiency – usually the most direct problems of users. The product produces failures and hence might not be available enough for

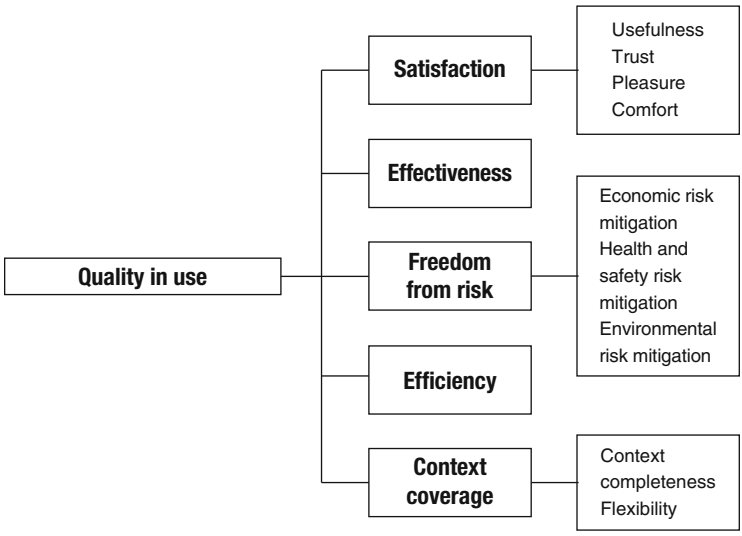

Fig. 2.10 Quality in use model of ISO/IEC 25010 [97]

the user. Of increasing importance is *security* as most systems are now accessible over networks. It includes to keep data intact and a secret as well as that the product needs to make sure that its users are who they claim to be. A characteristic aimed towards the developers is *maintainability* which describes that the system should be designed and programmed in a way that it is easy to comprehend, change and test. Finally, *portability* is also most important for developers who need to bring the product to another platform (programming language, operating system, hardware). It means that the necessary changes can be easily done and it can be easily installed.

2.3.3 Quality in Use Model

While the product quality model wants to describe the characteristics of the product directly, the quality in use model looks at characteristics of the interactions of different stakeholders with the product. The most prominent of those stakeholders is the primary user. This is why quality in use is often associated merely with usability. Quality in use, however, can also mean the quality in maintaining or porting the product or providing content for the product. Hence, to "use" and "user" have a very broad meaning in this context. The model describes the quality in use with five characteristics shown in Fig. 2.10.

The two core characteristics of quality in use are effectiveness and efficiency. *Effectiveness* is how well the product supports the user in achieving objectives, and *efficiency* denotes the amount of resources necessary to achieve these objectives. For drivers interacting with navigation systems in cars, it means: Can they get suitable

navigations to their desired destinations and how many times do they have to press a button for that? For maintainers, this is rather a maintenance task characterised by the number of new faults introduced and effort spent.

Quality in use goes beyond reaching your objectives with appropriate resources. The *satisfaction* of the user is also a quality characteristic that includes the trust the product inspires the users with and also the pleasure of using the product. In many contexts, also the *freedom from risk* is important. The most famous part thereof is the safety of systems that can harm humans. Environmental or economic risks play a role here as well. Finally, the standard also proposes that the usage should cover the appropriate context. *Context coverage* contains that the product understands the complete context of its usage and it can react flexible to changes in the context.

2.3.4 Summary and Appraisal

ISO/IEC 25010 provides with its two quality models a comprehensive list of relevant quality characteristics for software products and the interaction with these products. It is a useful structuring of the relevant topics and it can serve as a good checklist to help requirements engineers in not forgetting any quality characteristics and also quality engineers to analyse the quality of a system. In that, it is an improvement over the old ISO/IEC 9126, because it made some useful changes in the models, such as including security as a separate characteristic. It also reduced the number of quality models from three to two. It is still not clear, however, when it is advisable to use which model. It seems that most software companies employ only the product quality model and only when they analyse usability, the quality in use model is taken into account. The standard does not prescribe when to use which model apart from saying "The product quality model focuses on the target computer system that includes the target software product, and the quality in use model focuses on the whole human-computer system that includes the target computer system and target software product" [97]. It is important to understand, however, that these models are taxonomies. They describe one possible structuring of quality but in no way the only possible or sensible structuring. There will always be discussions about why a particular subcharacteristic is part of one characteristic and not the other. For example, is availability a subcharacteristic of reliability or security? In the end, for being a checklist, this does not matter. If you want to use the quality models beyond that, this simple structuring might not be sufficient. We will look into a more generic and more complete quality modelling framework in the next section.

2.4 Quamoco Quality Models

Already before the SQuaRE series and in parallel to its development, there had been research into improving ISO/IEC 9126 as the standard means to model and evaluate software product quality. Various research groups and companies worked

on quality models and how they can be applied best. One of the larger initiatives was the three-year German project *Quamoco*[7] sponsored by the German ministry for research and education. It brought together experts from industry (Capgemini, itestra, SAP AG and Siemens) and research (Fraunhofer IESE and Technische Universität München). After it had finished at the beginning of 2012, it delivered generic quality model concepts, a broad base model and comprehensive tool support all available under an open source licence [211]. It is more generic than ISO/IEC 25010, but it can be used in a way conforming to the ISO standard. We look in detail into the core results in the following.

2.4.1 Usage of Quality Models

Most commonly, we find quality models reduced to mere reference taxonomies or implicitly implemented in tools (see Sect. 2.1). As explicit and living artefacts, however, they can capture general knowledge about software quality, accumulate knowledge from their application in projects and allow defining a common understanding of quality in a specific context [51, 82, 138, 143].

In contrast to other quality models that are expressed in terms of prose and graphics only, our quality model is integrated into the software development process as basis of all quality assurance activities. As Fig. 2.11 shows, the model can be seen as project- or company-wide quality knowledge base that centrally stores the definition of quality in a given context. Experienced quality engineers are still needed for building the quality models and enforcing them with manual review activities. They can rely on a single definition of quality, however, and are supported by the automatic generation of guidelines. Moreover, quality assessment tools like static analysers, which automatically assess artefacts, and test cases can be directly linked to the quality model and do not operate isolated from the centrally stored definition of quality. Consequently, the quality profiles generated by them are tailored to match the quality requirements defined based on the model. We refer to this approach as *model-based quality control*.

The execution of this approach should be continuously in a loop. In the quality control loop [48], which we will discuss in detail in Sect. 4.1, the quality model is the central element for identifying quality requirements, planning quality assurance, assessing the quality requirements using the quality assurance results and reworking the software product based on the assessment results. The quality model is useful for defining what we need to measure and how we can interpret it to understand the state of the quality of a specific product. A single source of quality information avoids redundancies and inconsistencies in various quality specifications and guidelines. Furthermore, if there is a certification authority, we can also certify a product against

[7] http://www.quamoco.de.

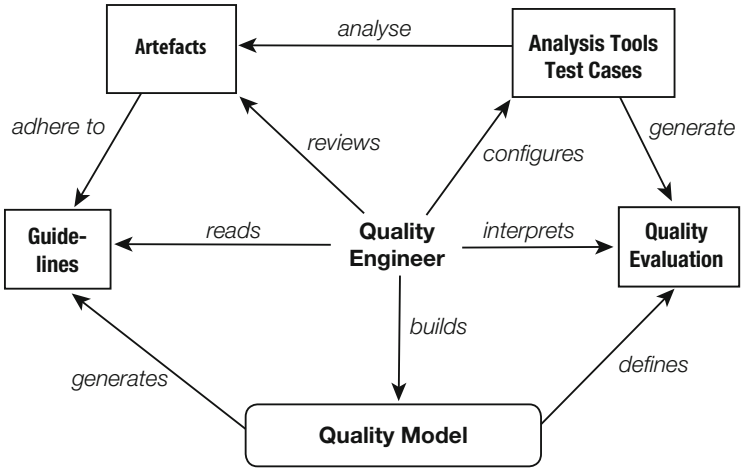

Fig. 2.11 Model-based quality control [46]

the quality model. This would be useful, for example, if we included guidelines from standards such as ISO 26262 [105] or Common Criteria [43].

On top of that, the model itself helps us to establish suitable and concrete quality requirements. The quality model contains quality knowledge that we need to tailor to the product to be developed. This includes removing unneeded quality characteristics as well as adding new or specific quality factors.

2.4.2 Concepts

The previous work of all Quamoco partners on quality models, our joint discussions and experiences with earlier versions of the meta-model brought us to the basic concept of a *factor*. A factor expresses a *property* of an *entity* which is similar to what Dromey [54] calls *quality carrying properties* of *product components*. We use *entities* to describe the things that are important for quality and *properties* for the attributes of the things we are interested in. Because the concept of a factor is general, we can use it on different levels of abstraction. We have concrete factors such as *cohesion of classes* as well as abstract factors such as *portability of the product*. To illustrate the following concepts, we show their relationships in Fig. 2.12.

To clearly describe quality from an abstract level down to concrete measurements, we differentiate the two factor types *quality aspects* and *product factors*. Both can be refined into sub-aspects and sub-factors, respectively. The quality aspects express abstract quality goals, for example, the quality characteristics of ISO/IEC 9126 and ISO/IEC 25010 which would have the complete product as their entity. The product factors are measurable attributes of parts of the product. We require the leaf product factors to be concrete enough to be measured.

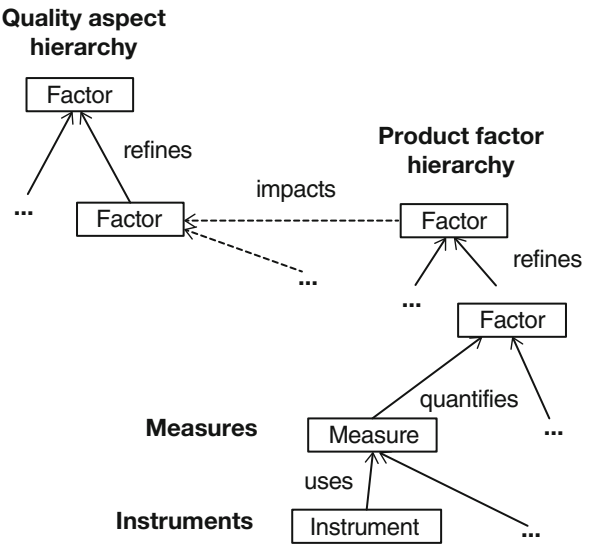

Fig. 2.12 The Quamoco quality model concepts

An example is the *duplication of source code* which we measure with *clone coverage*[8] (see Sect. 4.4.2). This clear separation helps us to bridge the gap between the abstract notions of quality and concrete implementations. In addition, separating the entities from their properties addresses the problem of the difficult decomposition of quality characteristics for product factors. For example, the entity *class* can be decomposed straightforwardly into its attributes and methods. Furthermore, the entity concept is abstract enough to model processes or people as entities which could also have quality-carrying properties.

Moreover, we are able to model several different hierarchies of quality aspects to express divergent views on quality. Quality has so many different facets that a single quality factor hierarchy is not able to express it. Also in ISO/IEC 25010, there are two quality hierarchies: product quality and quality in use. We can model both as quality aspect hierarchies, and other types of quality aspects are also possible. For example, we experimented with activity-based quality models [51] (similar to quality in use of ISO 25010) and technical classifications [175]. We found that quality aspects give us the flexibility to build quality models tailored for different stakeholders. In principle, the concept also allows us to have different product factor hierarchies or more levels of abstraction. In our experiences with building quality models, however, we found the two levels *quality aspect* and *product factor* to be sufficient.

To completely close the gap between abstract quality characteristics and assessments, we need to put the two factor types into relation. The product factors have *impacts* on quality aspects. This is similar to variation factors which have impacts on

[8]Clone coverage is the probability that a randomly chosen line of code is duplicated.

quality factors in GQM abstraction sheets [191]. An impact is positive or negative and describes how the degree of presence or absence of a product factor impacts a quality aspect. This gives us a complete chain from measured product factors to impacted quality aspects, and vice versa.

We need product factors that are concrete enough to be measured so that we can close the abstraction gap. Hence, we have the concept of *measures* for product factors. A measure is a concrete description of how a specific product factor should be quantified for a specific context. For example, this could be the number of deviations of a rule for Java such as that strings should not be compared by "==". A factor can have more than one measure if we need multiple measures to cover the concept of the product factor.

Moreover, we separate the measures from their *instruments*. The instruments describe a concrete implementation of a measure. In the example of the string comparison, an instrument is the corresponding rule as implemented in the static analysis tool *FindBugs*. This gives us additional flexibility to collect data for measures manually or with different tools in different contexts. Overall, the concept of a measure also contributes to closing the gap between abstract quality factors and concrete software, as there is traceability from quality aspects via product factors to measures and instruments.

Having these relationships with measures and instruments, it is straightforward to assign evaluations to factors so that we can aggregate from the measurement results (provided by the instruments) to a complete quality evaluation. There are different possibilities for implementing this. We will describe a detailed quality evaluation method using these concepts in Sect. 4.2. Moreover, we can go the other way round. We can pick quality aspects, for example, ISO/IEC 25010 quality characteristics that we consider important and costly for a specific software system and trace what product factors affect them and what measures quantify them [203]. This allows us to concentrate on the product factors with the largest impact on these quality aspects. It also gives us the basis for specifying quality requirements, what we will discuss in detail in Sect. 3.1.

Building quality models in such detail results in large models with hundreds of model elements. Not all elements are important in every context and it is impractical to build a single quality model that contains all measures for all relevant technologies. Therefore, we introduced a modularisation concept which allows us to split the quality model into *modules*. Modularisation enables us to choose appropriate modules and extend the quality model by additional modules for a given context. The *root* module contains general quality aspect hierarchies as well as basic product factors and measures. In additional modules, we extend the root module for specific technologies and paradigms, such as object orientation; programming languages, such as C#; and domains, such as embedded systems. This gives us a flexible way to build large and concrete quality models that fit together, meaning they are based on the same properties or entities. As basis for all specialised quality models, we built the *base model*, a broad quality model with the most common and important factors and measures, to be applicable to (almost) any software. We will describe the base model in more detail in Sect. 2.4.6.

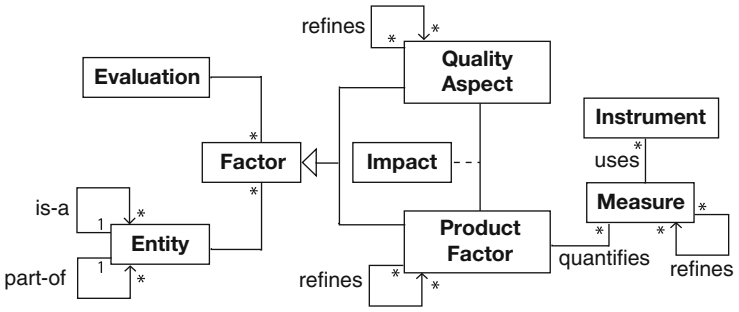

Fig. 2.13 The Quamoco quality meta-model

2.4.3 Meta-Model

We specified the general concepts described so far in a meta-model to make it precise. The benefit of an explicit meta-model is twofold: First, it ensures a consistent structure of quality models. Second, it constitutes a necessary basis for modelling tool support. The core elements of the meta-model are depicted as an (abstracted) UML class diagram in Fig. 2.13. Please note that we left out a lot of details such as the IDs, names and descriptions of each element to make the diagram more comprehensible. The central element of the meta-model is the *Factor* with its specialisations *Quality Aspect* and *Product Factor*. Both can be refined and, hence, produce separate directed acyclic graphs. An *Impact* can only exist between a *Product Factor* and a *Quality Aspect*. This represents our main relationship between factors and allows us to specify the core quality concepts.

A *Factor* always has an associated *Entity* which can be in an is-a as well as a part-of hierarchy. An Entity of a Factor can, in principle, be any thing, animate or inanimate, that can have an influence on software quality, e.g. the source code of a method or the involved testers. As we concentrate on product quality, we only choose product parts as entities for the product factors. We further characterise these entities by properties such as STRUCTUREDNESS or CONFORMITY to form a product factor. For the quality aspects, we often use the complete product as the entity to denote that they usually do not concentrate on parts of the product. The property of an *Entity* that the *Factor* describes is expressed in the *Factor*'s name.

Each *Factor* also has an associated *Evaluation*. It specifies how to evaluate the *Factor*. For that, we can use the evaluation results from sub-factors or – in the case of a *Product Factor* – the values of associated *Measure*s. A *Measure* can be associated with more than one *Product Factor* and has potentially several instruments that allow us to collect a value for the measure in different contexts, e.g. with a manual inspection or a static analysis tool.

We modelled this meta-model with all details as an EMF[9] model which then served as the basis for a quality model editor (see Sect. 2.4.8).

2.4.4 Product Entities and Product Factors

The major innovation and also the most useful new construct in the Quamoco quality models is the product factor. It is the centre of the quality model as it is measured and describes impacts on quality aspects and, thereby, the product's quality overall. Product factors are far more tangible than quality aspects as they always describe a concrete product part, an entity. This relationship to entities also allows us an hierarchical decomposition of product factors.

The organisation of entities in a hierarchy is mostly straightforward, because entities often already have hierarchical relationships. For example, methods are part of a class or an assignment is a kind of statement. In principle, there could be more complex relationships instead of hierarchies, but modelling and finding information tends to be easier if such complex relationships are mapped to hierarchies.

The top level is the complete Product which represents the root for all entities of the system. In principle, Quamoco quality models can describe more than the product. As we concentrate on product quality in this book, however, we focus on product entities only. In the example in Fig. 2.14, the product contains the three artefacts: source code, its user interface and its requirements. Again, these entities need to be further refined. For example, the source code could consist of functions as well as procedures – depending on the programming language.

The associations between entities in the entity hierarchy can have one of two different meanings: Either an entity is a part or a kind of its super-entity. Along the inheritance associations, properties could also be inherited. In our current tool support, however, we do not support inheritance because it made the relationships in the model more complex. In principle, the inheritance would allows us a more compact description and prevents omissions in the model. For example, naming conventions are valid for all identifiers no matter whether they are class names, file names or variable names. We made some experiences with that (Sect. 5.2) earlier and will look into including it again in the future.

A product factor uses these entities and describes their properties. In essence, we model the product factors similar as in an hierarchical quality model (Sect. 2.1). In addition, to make the quality models useful in practice, we always want concrete product factors that we can measure and evaluate. The granularity of the factors shown in the diagrams above is too coarse to be evaluated. Therefore, we follow the hierarchical approach in the product factors and break down high level factors into detailed, tangible ones which we call *atomic* factors. An atomic factor is a factor that

[9]Eclipse Modeling Framework, http://emf.eclipse.org/.

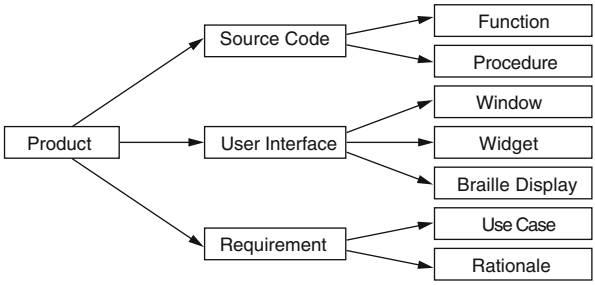

Fig. 2.14 Example entities

can or must be evaluated without further decomposition either because its evaluation is obvious or there is no known decomposition.

We found that a fine-granular decomposition of the product factors inevitably leads to a high number of repetitions as the same properties apply to different kinds of artefacts. For example, *consistency* is required for identifier names as well as for the layout of the documentation. This is the reason, we explicitly introduced properties such as STRUCTUREDNESS or CONFORMITY to characterise entities. The difference is that entities "are the objects we observe in the real world" and properties are "the properties that an entity possesses" [123]. The combination of an entity and a property forms a *product factor*. We use the notation (introduced in [46]) [Entity | PROPERTY] when we need to refer to a product factor in the text. For the example of code clones, we write [Method | REDUNDANCY] to denote methods that are redundant.

An influence of a product factor is specified by an impact which can be positive or negative. Using the notation introduced for product factors, we can express the impact a product factor has on a quality aspect with a three-valued scale where "+" expresses a positive and "−" a negative impact (the non-impact is usually not made explicit):

$$\text{[Entity e | PROPERTY P]} \xrightarrow{+/-} \text{[Quality Aspect a]}$$

Examples are [Debugger | EXISTENCE] $\xrightarrow{+}$ [Fault Diagnostics] which describes that the existence of a debugger has a positive influence on the activity fault diagnostics. [Identifier | CONSISTENCY] $\xrightarrow{+}$ [Analysability] describes that consistently used identifier names have a positive impact on the analysability of the product. As another example, [Variable | SUPERFLUOUSNESS] $\xrightarrow{-}$ [Code Reading] denotes that unused variables hamper the reading of the code. To overcome the problem of unjustified quality guidelines and checklists each impact is additionally equipped with a detailed description.

Note that the separation of entities and properties does not only reduce redundancy but allows us a clear refinement of the product factors. Let us illustrate that by an example of the quality taxonomy defined in [168]: *System Complexity*. As *System Complexity* appears too coarse-grained to be assessed directly, it is desirable to further decompose this element. The decomposition is difficult, however, as the decomposition criterion is not clearly defined, i.e. it is not clear

what a subelement of *System Complexity* is. A separation of the entity and the property as in [System | COMPLEXITY] allows for a cleaner decomposition as entities themselves are not valued and can be broken up in a straightforward manner, e.g. in [Subsystem | COMPLEXITY] or [Class | COMPLEXITY].

2.4.5 Measures and Instruments

For the purpose of defining quality, product factors, quality aspects and impacts between them are sufficient. Often, however, we want to grasp quality more concretely and prescribe detailed quality requirements or evaluate the quality of a software system. Then we specify measures that evaluate each atomic factor either by automatic measurement or by manual review. For higher-level statements about the quality of a system, we need to aggregate the results for these atomic measurements. The aggregation can be done along the refinement hierarchies of the product factors, quality aspects, and along the impacts. How you exactly perform the aggregation depends on your evaluation goal. We will discuss a detailed and tool-supported evaluation method in Sect. 4.2.

The product factors are the core elements of the model that need to be evaluated to determine the quality of a product. For this evaluation, we provide explicit measures for each factor. Since many important factors are semantic in nature and therefore not assessable in an automatic manner, we distinguish three measure categories:

1. Measures that we can collect with a tool. An example is an automated check for `switch` statements without a `default` case which could measure the product factor [Switch Statement | COMPLETENESS].
2. Measures that require manual activities, e.g. reviews. An example is a review activity that identifies the location and number of improper use of data structures for the corresponding [Data Structures | APPROPRIATENESS].
3. The combination of measures that we can assess automatically to a limited extent requiring additional manual inspection. An example is redundancy analysis where we detected cloned source code with a tool, but other kinds of redundancy must be left to manual inspection ([Source Code | REDUNDANCY]).

When we include measures in the quality model, it becomes similar to the result of the Goal-Question-Metric (GQM) approach. Another way of looking at Quamoco quality models is therefore as GQM patterns [12, 135]. The quality aspect defines the goal and the product factors are questions for that goal which are measured by certain metrics in a defined evaluation. For example, the goal is to evaluate modifiability which is analysed by asking the question "How consistent are the identifiers?" which in turn is part of an inspection.

We further distinguish *instruments* from *measures*. Measures are the conceptual description of a measurement, while an instrument is the concrete way for collecting data for that measure. For example, the product factor about complete switch

statements [Switch Statement | COMPLETENESS] has a measure *Missing default case* which counts the number of switch statements without a default case. For Java, we have an automatic instrument: The static analysis tool FindBugs has the rule *SF_SWITCH_NO_DEFAULT* that returns findings suitable for the measure. This distinction allows us to describe measures to some degree independent of the used programming language or technology and to provide different instruments depending on the available tools. For example, if no tools are used, the switch statements could also be checked in a manual review which would be a manual instrument.

2.4.6 Quality Aspects: Product Quality Attributes

The traditional and well-established way to describe quality is using a quality taxonomy of quality attributes or quality characteristics. As discussed earlier, this approach goes back to the first quality models of Boehm et al. [25] and McCall and Walters [148] from the 1970s. The most recent incarnation of this approach is defined in the product quality model of ISO/IEC 25010 (see Sect. 2.3). Therefore, the most straightforward quality aspect hierarchy is to directly use the quality characteristics defined there. The quality characteristics of ISO/IEC 25010 become quality aspects, and instead of assigning directly measure elements to the quality characteristics, as defined in the standard, we specify measurable product factors with impacts onto those quality characteristics. We can see that as an implementation of the quality property concept of the standard.

Therefore, we have quality aspects such as *reliability*, *usability* or *maintainability*. They have the corresponding sub-aspects as they have quality sub-characteristics in ISO/IEC 25010: The sub-aspects of *reliability* are *maturity*, *availability*, *fault tolerance* and *recoverability*. The sub-aspects of *maintainability* are *modularity*, *reusability*, *analysability*, *modifiability* and *testability*. It is a simple transformation from the standard to the Quamoco meta-model. For all quality aspects, the corresponding entity is Product. Hence, the top-level quality aspect is [Product | QUALITY] for this quality aspect hierarchy. As the main information for this quality aspect hierarchy lies in the properties, i.e. the quality characteristics and attributes, we often omit Product as an abbreviation.

Let us look at three examples of product factors and their influence on this kind of quality aspects. The product factor [Subroutine | TERMINATION] describes "A method terminates if its execution ends in a finite time correctly". The product factor is measured by analyses of infinite recursions and infinite loops. If the product factor is present, it has a positive influence on the reliability of the product:

[Subroutine | TERMINATION] $\xrightarrow{+}$ [Reliability]

The product factor [Boolean expression | UNNECESSARILY COMPLICATED] is defined as "A boolean expression is unnecessarily complicated if the intended purpose could be met easier". This product factor is a refinement of the product factor [Source code part | UNNECESSARILY COMPLICATED]. This means that there are different source code

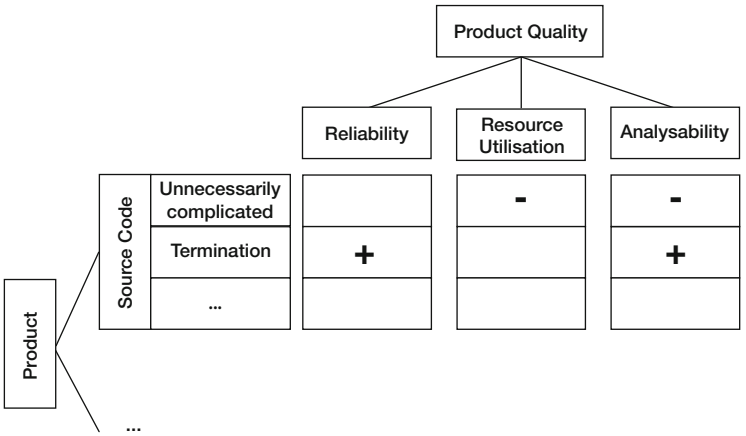

Fig. 2.15 Quality matrix for product quality attributes

parts that can be unnecessarily complicated in different ways. In this case, we have an influence on resource utilisation, which is a sub-aspect of performance efficiency, as the Boolean expression needs more memory than actually necessary:
[Boolean expression | UNNECESSARILY COMPLICATED] $\xrightarrow{-}$ [Performance efficiency]

The same product factor influences also analysability, which is a sub-aspect of maintainability, because the code is more complex and, hence, harder to understand:
[Boolean expression | UNNECESSARILY COMPLICATED] $\xrightarrow{-}$ [Analysability]

Figure 2.15 shows an excerpt of the quality model using product quality characteristics as quality aspects visualised as a matrix. It depicts the decomposition of the product into the source code and corresponding product factors. The "+" and "–" show a positive or negative impact on a part of the quality aspect hierarchy.

2.4.7 Quality Aspects: Activity Based or Quality in Use

A way to make the abstract and multifaceted concept of quality more concrete is to map it to costs: the cost of quality (Sect. 1.3). Yet, it is also difficult to find and quantify these costs. We were inspired by the concept of *activity-based costing* that uses activities in an organisation to categorise costs. It enables us to reduce the large category of indirect costs in conventional costing models by assigning costs to activities. Why should this assigning of costs to activities not work for quality costs? As a consequence, we introduced activities as first-class citizens into quality models [46, 51].

Besides, we were also motivated from our experiences with building large hierarchical quality models for maintainability. In this process it became harder and harder to maintain a consistent model that adequately describes the interdependencies

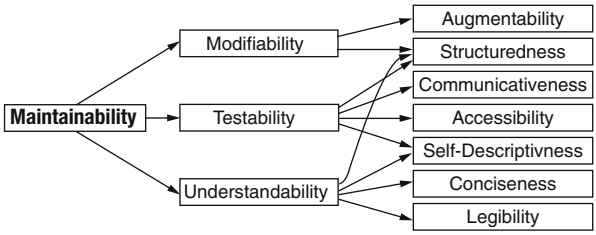

Fig. 2.16 Software Quality Characteristics Tree

between the various quality criteria. A thorough analysis of this phenomenon revealed that our model, and indeed most previous models, mixed nodes of two different kinds: maintenance *activities* and *characteristics* of the system to maintain. An example of this problem is the *maintainability* branch of Boehm's *Software Quality Characteristics Tree* [25] in Fig. 2.16.

Though (substantivated) adjectives are used as descriptions, some nodes refer to activities (modify, test, understand), whereas others describe system characteristics (structuredness, self-descriptiveness), albeit very general ones. So the model should rather read as: When we *maintain* a system we need to *modify* it and this activity of *modification* is (in some way) influenced by the *structuredness* of the system. While this difference may not look important at first sight, we found that this mixture of activities and characteristics is at the root of the decomposition problem encountered with previous models. The semantics of the edges of the tree is unclear or at least ambiguous because of this mixture. Since the edges do not have a clear meaning, they neither indicate a sound explanation for the relation of two nodes nor can we use them straightforwardly to aggregate values.

As the actual maintenance efforts strongly depend on both, the type of system and the kind of maintenance activity, it is necessary to distinguish between activities and characteristics. For example, consider two development organisations of which company A is responsible for adding functionality to a system while company B's task is merely fixing bugs of the same system just before its phase-out. One can imagine that the success of company A depends on different quality criteria (e.g. architectural characteristics) than company B's (e.g. a well-kept bug tracking system). While both organisations will pay attention to some common characteristics such as a good documentation, A and B would and should rate the maintainability of the system in quite different ways, because they are involved in fundamentally different *activities*. Looking at maintainability as the productivity of the maintenance activity within a certain context widens the scope of the relevant criteria. A and B's productivity is determined not only by the system itself but also by a plethora of other factors, which include the skills of the engineers, the presence of appropriate software processes and the availability of proper tools like debuggers. Therefore, our quality models are in principle not limited to the software system itself, but we can describe the whole *situation* [51]. In this book, however, we focus on product quality and, hence, will consider only *product factors*.

Based on these insights, we proposed activity-based quality models (ABQM) to address the shortcomings of existing quality models [51]. They use the idea of avoiding high-level "-ilities" for defining quality and instead break quality down into detailed factors and their influence on activities performed on and with the system. Over time, we have found that ABQMs are applicable not only to maintainability but also to all quality factors. We have built concrete models for maintainability [51], usability [217] and security (see Sect. 2.6.2). The basic elements of ABQMs were also product factors – characteristics of the system – together with a justified impact on an activity.

Therefore, we can use activities as a quality aspect hierarchy. An entity of an activity as quality aspect can be any activity performed on and with the system such as *modification* or *attack*. For activities, we often omit an explicit property and assume we mean effectiveness and efficiency of the activity. This also fits well with the quality in use model of ISO/IEC 25010 that describes different usages of the system where *use* can include, for example, maintenance. Hence, uses from ISO/IEC 25010 are equivalent to activities from ABQMs.

Let us consider the example of a web shop written in Java. If there are redundant methods in the source code, also called clones (see Sect. 4.4.2), they exhibit a negative influence on *modifications* of the system, because changes to clones have to be performed in several places in the source code:

[Method | REDUNDANCY] $\xrightarrow{-}$ [Modification]

Another important activity is the *usage* of the system which is, for example, influenced by empty *catch* blocks in the code which show that exceptions are not handled:

[Catch Block | EMPTINESS] $\xrightarrow{-}$ [Modification]

We can again visualise the separation of quality aspects and product factors as a two-dimensional quality model. As top level, we use the activity Activity which has sub-activities such as Use, Maintenance or Administration. These examples are depicted in Fig. 2.17. The product hierarchy is shown on the left and the activity hierarchy on the top. The impacts are entries in the matrix where a "+" denotes a positive and a "−" a negative impact.

The matrix points out what activities are affected by which product factors and allows us to aggregate results from the atomic level onto higher levels in both hierarchies because of the unambiguous semantics of the edges. Hence, one can determine that concept location is affected by the names of identifiers and the presence of a debugger. Vice versa, cloned code has an impact on two maintenance activities. The example depicted here uses a Boolean relation between factors and activities and therefore merely expresses the existence of a relation between a factor and an activity. To express different directions and strengths of the relations, you can use more elaborate scales here. For quality evaluations, we can further refine this to a numerical scale (see Sect. 4.2).

The aggregation within the two hierarchies provides a simple means to cross-check the integrity of the model. For example, the sample model in Fig. 2.17 states that tools do not have an impact on coding which is clearly nonsense. The problem

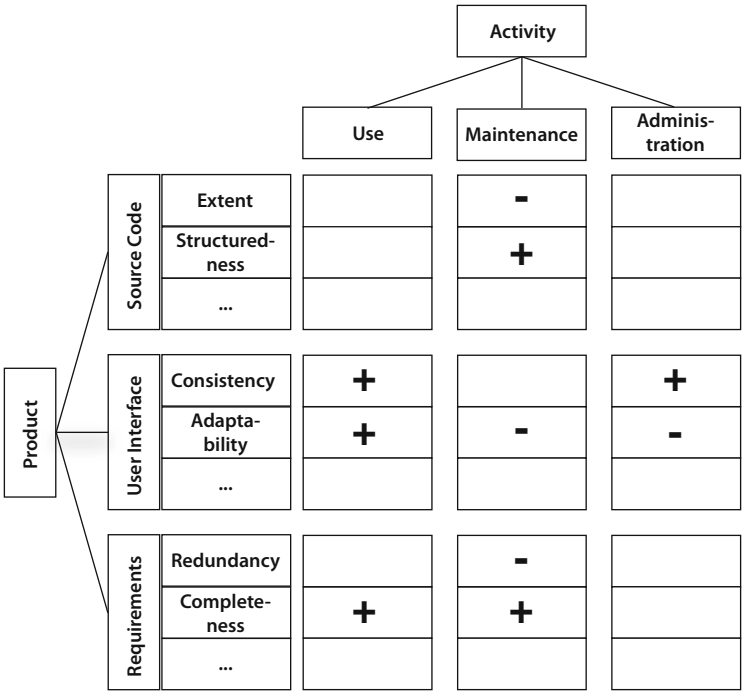

Fig. 2.17 Example quality matrix

lies in the incompleteness of the depicted model that does not include tools like integrated development environments.

As we mentioned above, we often omit explicit properties for activities where it is clear that we mean the effectiveness and efficiency of the activities. Nevertheless, in some areas of the quality model, it is useful to have the possibility to be more precise. For example, when we describe usability aspects, we specify properties of the usage activity such as *satisfaction* or *safety* [217]. Then we can analyse the different aspects of the usage of the system more accurately. In many cases, however, it is more interesting to find a suitable decomposition of the relevant activities and only consider effectiveness and efficiency of those.

2.4.8 Tool Support

Comprehensive quality models contain several hundred model elements. For example, the maintainability model that was developed for a commercial project in the field of telecommunication [27] has a total of 413 model elements consisting of 160 product factors (142 entities and 16 properties), 27 activities and 226 impacts. Hence, quality models demand a rich tool set for their efficient creation, management and application just like other large models, e.g. UML class diagrams.

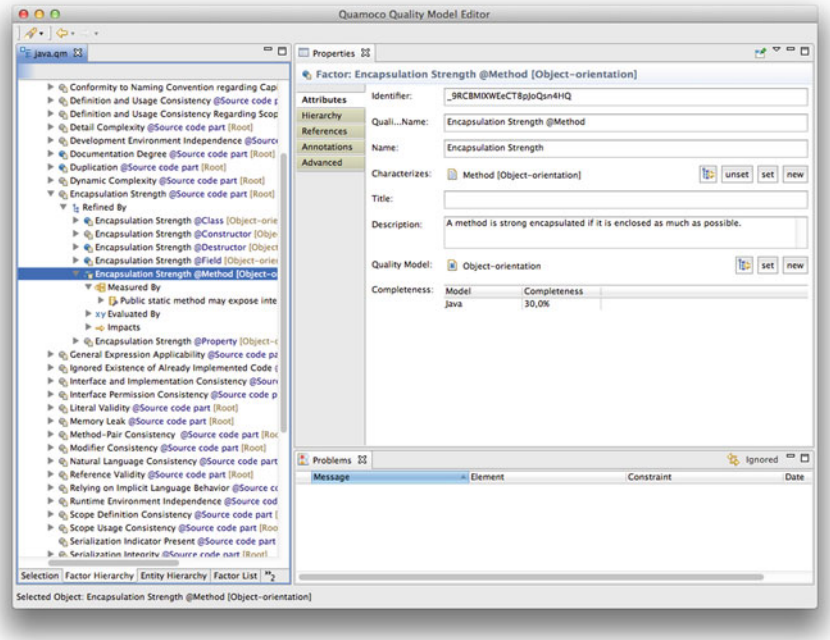

Fig. 2.18 The Quamoco quality model editor

Because our quality models are based on an explicit meta-model, we are able to provide a model editor that does not only allow the initial development of quality models but also supports other common tasks like browsing, persistence, versioning and refactoring (Fig. 2.18). In Quamoco, we have developed a tool to define this kind of large and detailed quality models. Besides the easier creation and modification of the model, this has also the advantage that we can automate certain quality assurance tasks. For example, by using the tool we can automatically generate customised review guidelines for specific views. In addition, we have implemented a direct coupling with the quality dashboard tool ConQAT[10] so that the results of diverse quality analysis tools, tests and reviews can be read and aggregated with respect to the quality model.

One of the most powerful features of the model editor is the automatic generation of guideline documents from the quality model. This enables us to transfer the abstract definition of quality stored in the model to a format developers are familiar with. Moreover, unlike classic, handwritten guidelines, the automatically generated ones are guaranteed to be synchronised with the quality model that explicitly

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

captures the understanding of quality within a project or a company. You can tailor guideline documents for specific needs by defining selected views on the model. For example, a guideline document could be specifically generated for documentation review sessions.

2.4.9 Summary

The Quamoco approach to modelling quality advances on previous approaches to model quality in the following issues:

Flexibility in the Quality Aspects

The Quamoco quality modelling approach allows to completely include the standard ISO/IEC 25010 quality models. We can use product quality characteristics or quality in use activities as quality aspects. If we need other views on the product factors, it would be no problem to include other quality aspect hierarchies.

Unambiguous Decomposition Criteria

Previous hierarchical models often exhibit a "somewhat arbitrary selection of characteristics and sub-characteristics" [121,122]. We claim this is because previous models lack a clearly defined decomposition criterion. For example, it is not clear how *self-descriptiveness* relates to *testability* in Boehm's quality characteristics tree (Fig. 2.16) if one is not satisfied with a trivial "has something to do with". The activity-based approach overcomes this shortcoming by separating aspects that are often intermingled: activities, entities and properties. This separation creates separate hierarchies with clear decomposition criteria.

Explicit Meta-Model

Unlike most other approaches known to us, our approach is based on an explicitly defined meta-model. This enables us to provide a rich set of tools for editing and maintaining quality models. Most important, the meta-model is a key for the model-based quality control approach outlined before. Additionally, the meta-model fosters the conciseness, consistency and completeness of quality models as it forces the model builders to stick to an established framework and supports them in finding omissions.

Complete Model

The most important advantage of Quamoco quality models is the complete chain from abstract quality aspects over product factors to concrete measurements. This allows us to define comprehensible evaluation methods as well as to refine quality requirements to be concrete and measurable.

2.5 Quality Model Maintenance

The quality model that defines the quality requirements of a system is, like the system itself, not a static entity. On the contrary, it needs to be updated and adapted over time. I will describe sources for model changes and give methodological guidelines for model maintenance.

2.5.1 Sources for Model Changes

As in software, changes of the model can come from various sources. Most often, changes in the used technology, programming language or framework induce changes in the model. The providers of these technologies introduce new concepts, new language constructs or new libraries which all have their own product factors that we need to cover in the quality model.

In addition, the model is not fault free. It is built by humans and therefore there are incorrect model elements or relationships between model elements. For example, there might be an impact from the conciseness of identifiers in the code to the understanding of the end user. This is a mistake of the modeller who mixed up the activities. We need to correct this kind of faults. In the same example, the modeller probably then forgot to include an impact to the understanding of the developer. We also need to correct these omissions.

A different kind of omission is, if we find new product factors. Empirical research aims at finding such factors that significantly influence different quality factors. A new study might find that the level of redundancy in the source code has an influence on the number of faults in the system and, thereby, it has an impact on how often an end user experiences failures. We need to change the model to include such new factors. Similarly, empirical research or experience in a company can improve the knowledge about a product factor, e.g. what is the important property that influences quality or how strong is the impact. These refined informations also lead to changes in the model.

Finally, you can also change the quality goals and thereby induce a change in the model. If you realise that your system has become security-critical over time, you can create corresponding quality goals. To fulfil these goals, we need

to extend the quality model with product factors that impact the quality aspects related to these goals. Analogously, quality goals can decrease in importance and be a reason to remove certain quality factors from the model.

2.5.2 Analysis and Implementation

A model change is similar to a software product change. First, we need to analyse the change request and then implement it in the model. The analysis determines what is wrong or missing in the model, which elements of the quality model are affected. An entity, a property or a whole product factor can be affected. For example, its name or description might be misleading. The relationships between factors can be wrong or missing. For example, an impact between two factors might be missing.

After the analysis, change the model either by changing element names, descriptions or additional information, by adding new elements or by deleting existing elements. It is important that you check the consistency of the model after each change. Removing model elements might affect other parts of the model that have then impacts for which the target is missing, for example. A good model editor can help you in showing you warnings for actual or potential inconsistencies.

2.5.3 Test

After you performed all changes and checked that the model is consistent again, you need to test if your model still works correctly in your quality control loop. This is important, because a quality model consists of a lot of text that carries semantics. Consistency and correctness of these texts cannot be checked automatically. Also evaluations that you specified can only be checked syntactically. If they make sense is still open. For that, we need to perform tests in the control loop.

It is useful to have a standard test system that you evaluate with the quality model after you performed changes on the model. The test system should not be different before and after the model change. If you keep the evaluation results of each test run, you can compare the results. There should be differences, because you changed the model. The differences, however, should only be the ones you expected by changing the model. If any unexpected differences occur, check if they are reasonable for the test system or if you introduced new faults into the model. Problems that you detect here become new change requests for the model and lead to re-analysis and re-modelling.

In case you worked on descriptions of model elements that you also use to generate review checklists and guidelines, a useful test is also to give it to several developers and testers and ask them for feedback. They are the ones who need to understand what you wanted to say with the description. Make sure they really understand.

2.5.4 Checklist

- Do you regularly check relevant publications for new product factors or improved knowledge about product factors?
- Do you feed back problems with the model that you found in the quality control loop into model changes?
- Do you regularly reevaluate your quality goals and change your quality model accordingly?
- Before you perform changes, have you analysed which model elements are affected?
- After you had performed a change, have you checked the model's consistency?
- Do you have a test system for which you keep the evaluation results?
- For each model change, do you perform test runs on test systems?
- For changes in checklists and guidelines, do you ask developers and testers for feedback?

2.6 Detailed Examples

We look in this section into three detailed examples of quality models to provide an illustration of the core concepts. The first example quality model follows directly the Quamoco approach. It is the performance efficiency part of the Quamoco base model. The second example uses the Quamoco concepts but only to describe quantifiable quality requirements without evaluation specifications. In particular, it uses an activity-based decomposition of attacks to describe security. The third example contrasts the other two, as it is a reliability growth model that illustrates a very focused prediction model.

2.6.1 Performance Efficiency Part of the Quamoco Base Model

The Quamoco base model is one of the main results of the Quamoco project. It captures the experiences and discussions of all experts who took part in the project. The whole base model is too large to describe here, but it is available on the Quamoco web site.[11] Our aim here is to illustrate the quality model concepts, and therefore, we concentrate on one smaller part of the model: the performance efficiency model with measurements for Java.

Performance efficiency is the quality characteristic of ISO/IEC 25010 describing how well the software uses the resources for a fast execution. In the standard, it is

[11] http://www.quamoco.de.

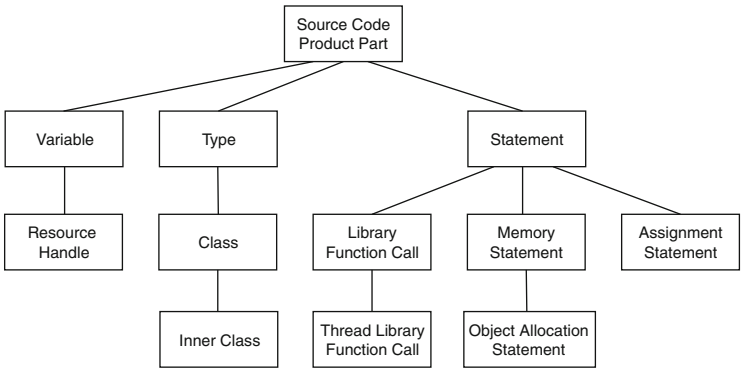

Fig. 2.19 An excerpt of the entity tree for the performance model

defined as "The performance relative to the amount of resources used under stated conditions" [97]. It is decomposed into two sub-characteristics: resource utilisation and time behaviour. Resource utilisation is "the amounts and types of resources used when the product performs its function under stated conditions in relation to an established benchmark", and time behaviour is "the response and processing times and throughput rates when performing its function, under stated conditions in relation to an established benchmark" [97]. We have not defined a benchmark in Quamoco but looked at statically visible problems in the source code that can impact these quality characteristics.

Figure 2.19 shows an excerpt of the entities tree of entities with an impact on performance efficiency or one of its sub-characteristics. You can see that various common source code parts have an influence, such as assignment statement, but several entities have special purposes that already suggest a relation to performance efficiency, such as resource handle or object allocation statement.

Using these and other entities, there are more than 50 product factors with measurements for Java alone influencing performance efficiency and its sub-characteristics. We will look at several examples that have a relatively strong impact according to our calibration of the base model.

The strongest impact on resource utilisation has the product factor describing resource handles: [Resource Handle | DEFINITION AND USAGE CONSISTENCY]. In the model, it is defined that "A resource handle's definition and usage are consistent if its definition allows usage in the intended way and if it is used in the defined way". The resource handles in Java are, for example, streams or databases. The corresponding measures for this product factor hence are static checks about closing and cleaning up streams or databases. FindBugs provides some rules to check for these kinds of bug patterns. We weighted all of them the same but use different linear decreasing functions depending on the number of warnings for each FindBugs rule.

A second example of a product factor is [Inner Class | UNNEEDED RESOURCE OVERHEAD]. Inner classes are useful for various purposes, and they normally keep in Java an embedded reference to the object which created it. If this is not used,

we could reduce the size of the inner class by making it static. These and similar related problems can create an unneeded resource overhead for inner classes, which is expressed by this product factor. Again, we use a set of FindBugs rules to measure the size of the problem for a given software.

A third example is [Object Allocation Statement | USELESSNESS] which we will use to discuss an evaluation in detail (see also Sect. 4.2). We defined this product factor in the quality model with "An object allocation statement is useless if the object is never used". For Java, this is measured using two measures which use FindBugs intstruments: *Exception created and dropped rather than thrown* and *Needless instantiation of class that only supplies static methods*. Each measure has a weight of 50 %. Both are modelled with linear increasing functions. The exception measure adds its 50 % if the density of exceptions created and dropped is above 0.00003 and the class instantiation measure if there is any such warning. Together they describe on a range from 0 to 1 to what degree there are useless object allocation statements in the software. This product factor influences resource utilisation and time behaviour (among others). As mentioned before, resource utilisation has more than 50 product factors influencing it of which [Object Allocation Statement | USELESSNESS] has a weight of 6.65 %. Together with the other product factors, this will give an evaluation of the quality aspect. You will find more details on the quality evaluation method in Sect. 4.2.

In summary, this model can only partially evaluate the inherently dynamic quality "performance efficiency". Nevertheless, we found over 50 product factors with several measures for each product factor that can be determined statically. This is an interesting aspect because it is far cheaper and easier than dynamically testing based on a benchmark. For exact performance efficiency analyses, however, we will need to add these dynamic analyses. For an early and frequent look onto this quality factor, however, this part of the Quamoco base model is suitable. In the spirit of COQUAMO, you can see it as a quality indicator during development.

2.6.2 Web Security Model

Malicious attacks on software systems are a topic with high public visibility as average users can be affected. The level of vulnerabilities is still high today. For example, the CERT[12] reported 6,058 total vulnerabilities for the first 9 months of 2008. These attacks have a strong financial impact. In an E-Crime Watch Survey,[13] it is stated that on average for each company security attacks result in a monetary loss of $456,700 in 12 months.

Therefore, software security is still a large and important problem in practice. It affects not only financial aspects but also ethical issues such as privacy. Hence, it is

[12] http://www.cert.org/stats/vulnerability_remediation.html.

[13] http://www.csoonline.com/documents/pdfs/e-crime_release_091107.pdf.

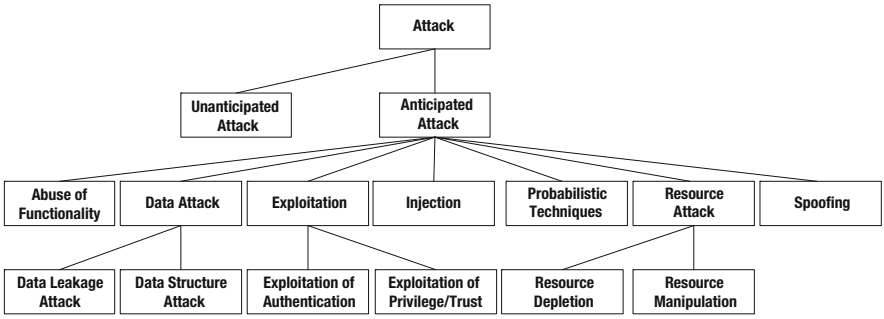

Fig. 2.20 Excerpt of the upper layers of the attack sub-tree of the activity hierarchy [214]

an important goal in software development to produce *secure* systems. This especially holds for web systems that are usually accessible in public networks. Secure web systems begin with the specification of security requirements. To become effective, they need to be clear and precise so that they are a real guidance for their implementation. This way, vulnerabilities can be prevented or at least reduced.

Despite the importance of high-quality security requirements for web systems, in practice they are often not well documented or not documented at all. This results also in often only poorly tested security properties of the final systems.

For specifying and verifying web security requirements, we created a web security instance of an activity-based quality model. Most important in this instance is to add attacks to the activity hierarchy. These are activities that need to be negatively influenced. First, we have to differentiate between anticipated attacks and unanticipated attacks. A major problem in software security is that it is impossible to know all attacks that the system will be exposed to, because new attacks are developed every day. Hence, to assure quality, we need to employ two strategies: (1) prepare the system against anticipated attacks and (2) harden the system in general to avoid other vulnerabilities. For the classification of the attacks, there are several possible sources. We rely on the open community effort *Common Attack Pattern Enumeration and Classification* (CAPEC) [83] that is led by the U.S. Department of Homeland Security. In the CAPEC, existing attack patterns are collected and classified. The attacks are organised in a hierarchy that is adapted for the activities hierarchy (Fig. 2.20).

The creation of the product factors is far more complicated. The product factors need to contain the available knowledge about characteristics of the system, its environment and the organisation that influence the described attacks. We employed various sources for collecting this knowledge including the ISO/IEC 27001 [106], the Web Guidelines,[14] OWASP [216] and the Sun Secure Coding Guidelines for the Java Programming Language [194]. However, two main sources were used

[14] http://www.webguidelines.nl/.

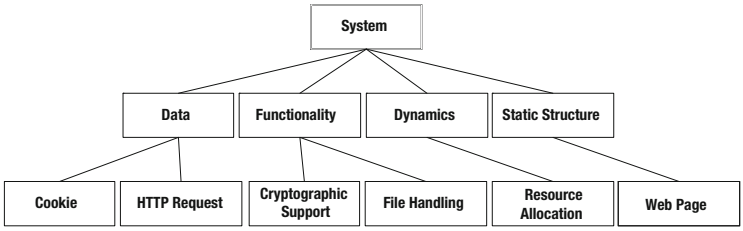

Fig. 2.21 Example entries of the system sub-tree from the entities [214]

because they constitute large community efforts and hence provide consolidated knowledge: specific parts of the *Common Criteria* (CC) [43] and the *Common Weakness Enumeration* (CWE) [84]. The Common Criteria describe requirements on a system that should ensure security with a focus on what the system "shall do". The CWE looks at security from the other direction and describes reoccurring weaknesses in software systems that lead to vulnerabilities that are exploited by attacks. Therefore, these two sources combined give a strong basis for the product factors.

We cannot describe the incorporation of the sources in all details in this example, but we give some instances on how knowledge from the sources has been modelled in our security model. For this, we use a sub-tree of the entity hierarchy for the system as depicted in Fig. 2.21. The system consists of Data and Functionality. Furthermore, it has Dynamics and a Static Structure. These entities have then again children. For example, data can be a Cookie or an HTTP Request. Interesting functionality can be Cryptographic Support or File Handling.

Many of the entries in the quality model that have their origin in the Common Criteria are modelled as a part of Functionality because they mainly describe behavioural aspects that nevertheless are important for security. An example that is shown in Fig. 2.21 is the cryptographic support of the system. Following the CC, this can be decomposed into Cryptographic Key Management and Cryptographic Operation. A further part of Cryptographic Key Management is the Cryptographic Key Generation. The CC defines a requirement for that key generation that it shall be in accordance with a specified algorithm and specified key sizes. In the model, we express that by using the property APPROPRIATENESS for Cryptographic Key Generation. The resulting factor [Cryptographic Key Generation | APPROPRIATENESS] is textually described by "The system generates cryptographic keys in accordance with a specified cryptographic key generation algorithm and specified cryptographic key sizes that meet a specified list of standards". Unfortunately, the CC does not contain any description of impacts. This would make the standard more useful because the motivation to use these requirements would be higher. Hence, we complete the information using other sources. In this case, the CAPEC contains possible solutions and mitigations in the description of the *cryptanalysis* attack that includes the recommendation to use proven cryptographic algorithms with recommended key sizes. Therefore, we include the corresponding negative impact of [Cryptographic Key Generation | APPROPRIATENESS] on Cryptanalysis.

In contrast to the CC, the *Common Weakness Enumeration* mainly provides characteristics of the system and especially the kind of code that should be avoided. We include these characteristics into the model in this negative way with a positive influence on the attacks, i.e. making attacks easier. Another possibility is to reformulate the weaknesses as strength that are needed with negative influence on attacks. We used both possibilities depending on which option was more straightforward to model.

Several weaknesses in the CWE are not aiming at specific attacks but describe characteristics that are indicators for possible vulnerabilities. We model these as factors that have an impact on unanticipated attacks. An example from the CWE that is in our security model is *dead code*. Several parts of the code can be superfluous such as variables, methods or complete classes. For a variable, we can model that as a positive impact of [Variable | SUPERFLUOUSNESS] on Unanticipated Attack.

2.6.3 Reliability Model

Reliability theory and models for estimating and predicting reliability have been developed for several decades. Reliability models are a very different type of quality model than the models we build with the Quamoco approach. They have a certain usage in practice and their application areas. We explain some basics and summarise the merits and limitations in the following. The general idea behind most reliability models is that we want to predict the future failure behaviour – and thereby the reliability – of a software based on actual failure data. Reliability is already defined as probability. That means that we use data from failures as sample data in a stochastic model to estimate the model parameters. There are other kinds of reliability models that use different mechanisms, but they are not as broadly used. The failure data that comprises the sample data can be one of two types: (1) time between failure (TBF) data or (2) grouped data. The first type contains each failure and the time that has passed since the last failure. The second type has the length of a test or operation interval and the number of failures that occurred in that interval. The latter type is not as accurate as the first, but the data is easier to collect in real-world projects.

Having used the sample data to estimate the model parameters, we can use the model with these parameters to predict future behaviour. That means the model can calculate useful quantities of the software at a point in time in the future. Apart from reliability the mostly used quantities are the expected number of failures up to a certain time t (often denoted by $\mu(t)$) and its derivative, the failure intensity (denoted by $\lambda(t)$). The latter can intuitively be seen as the average number of failures that occur in a time interval at that time t. Based on these quantities we can also predict, for example, for how long we have to continue testing to reach a certain failure intensity objective.

An excellent introduction to the topic is a book by Musa [158]. A wide variety of partly practical relevance is described in a handbook [139]. Finally, the most detailed description of these models and the theory behind them can be found in [159].

Execution Time and Calendar Time

Experience indicates that the best measure of time is the actual CPU execution time [159]. The reliability of software as well as hardware that is not executed does not change. Only when it is run there is a possibility of failure and only then there can be a change in reliability. The main cause of failure for hardware is considered the wear-out. Therefore, it is possible to relate the execution time to calendar time in some cases. For software this seems to be more difficult and hence execution time should be used.

However, CPU time may not be available, and it is possible to reformulate the measurements and reliability models in terms of other exposure metrics: clock time, in-service time (usually a sum of clock times due to many software applications running simultaneously on various single- or multiple-CPU systems), logical time (such as number of executed test cases, database queries or telephone calls) or structural coverage (such as achieved statement or branch coverage). One hundred months of in-service time may be associated with 50 months (clock time) of two systems or 1 month (clock time) of 100 systems. All of these approximations are also referred to as *usage time*.

In any case, some combination of statistical sampling with estimates of units sold is much better than using calendar time because of the so-called loading, or ramping, effect [111]. If this effect is not accounted for, most models assume a constant usage over time which is not reasonable under many practical circumstances. The difficulty is to handle differing amounts of installations and testing effort during the life cycle of the software.

Parameter Estimation

Parameter estimation is the most important part of applying a software reliability model to a real development project. The model itself is hopefully a faithful abstraction of the failure process, but only the proper estimation of the model parameters can fit the model to the current problem. There are several ways to accomplish that, but a consensus seems to be reached that a Maximum Likelihood approach on the observed failure data fits best to most models. Another possibility is to use other software metrics or failure data from old projects as basis for the estimation, but this is only advisable when no failure data is available, i.e. before system testing.

Merits and Limitations

The usage of stochastic reliability models for software is a proven practice and has been under research for decades now. Those models have a sound mathematical basis and are able to yield useful metrics that help in determining the current reliability level and in deciding the release problem.

However, the models are often difficult to use because (1) some understanding of the underlying statistics and (2) laborious, detailed metrics documentation and collection are necessary. Moreover, the available tool support is still not satisfying. The main deficiency is, however, that the models are only applicable during system test and field usage and even during system test they depend on usage-based testing, i.e. they assume that the tests follow the operational profile that mirrors the future usage in the field. The approach to use other metrics or failure data from old projects is very fragile. Especially to use other software metrics has not led to a satisfying predictive validity. This narrows the merit of those models.