TUM

TECHNISCHE UNIVERSITÄT MÜNCHEN
INSTITUT FÜR INFORMATIK

The Quamoco Quality Meta-Model

Stefan Wagner, Klaus Lochmann, Sebastian Winter, Florian Deissenboeck, Elmar Juergens, Markus Herrmannsdoerfer, Lars Heinemann, Michael Kläs

TUM-I128

The Quamoco Quality Meta-Model

Stefan Wagner*, Klaus Lochmann, Sebastian Winter*, Florian Deissenboeck, Elmar Juergens, Markus Herrmannsdoerfer*, Lars Heinemann
Institut für Informatik, Technische Universität München
Garching, Germany
{wagnerst,lochmann,winterse,deissenb,juergens,herrmama,heineman}@in.tum.de

Michael Kläs, Adam Trendowicz, Jens Heidrich
Fraunhofer IESE
Kaiserslautern, Germany
{michael.klaes,adam.trendowicz,jens.heidrich}@iese.fraunhofer.de

Reinhold Ploesch
Institut für Wirtschaftsinformatik—Software Engineering,
Johannes Kepler University Linz
Linz, Austria
reinhold.ploesch@jku.at

Andreas Goeb
SAP Research, Software Engineering & Tools
Darmstadt, Germany
andreas.goeb@sap.com

Christian Koerner
Siemens AG, Corporate Technology–SE 1,
Munich, Germany
christian.koerner@siemens.com

Korbinian Schoder, Jonathan Streit
itestra GmbH
Munich, Germany
{schoder, streit}@itestra.de

Christian Schubert
Capgemini Deutschland GmbH
Berlin, Germany
christian.schubert@capgemini.com

* At the time of writing the author has been with Technische Universität München.

Table of Contents

Table of Figures

1 Introduction

A quality model specifies meaning of quality of a software product in a way that it can be used in a number of scenarios, as, for example, to assess or to improve the quality of the software product. A quality metamodel defines the rules that Quamoco compliant quality models need to obey. This document describes the quality metamodel that is developed by Quamoco as the main result of Work Package 1.3.

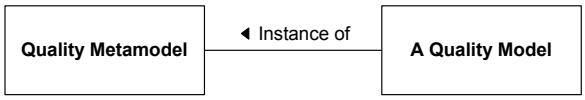

Figure 1: Relationship between the metamodel and a quality model.

Figure 1 depicts the relationship between the quality metamodel defined by this document, and a quality model that is an instance of the metamodel. A quality model is said to be an instance of the metamodel if it conforms to the structure defined by the metamodel. The metamodel is defined on the following two levels:

1. The *definition level* provides the concepts to define the qualities of a software product and how to assess them.
2. The application of a quality model to assess the quality of a software product generates plenty of data, for example, the measurement results and the results of the evaluations. The *application level* specifies the concepts to store this data.

2 Concepts of the Metamodel

This section provides a brief overview of the concepts and relationships provided by the quality metamodel.

2.1 Definition Level

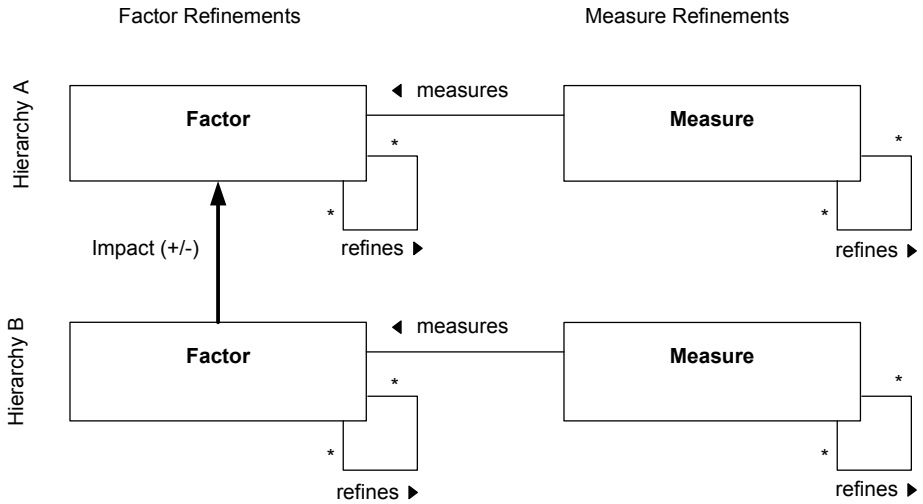

Figure 2: Basic concepts of the metamodel on the definition level.

Figure 2 complemented by Figure 3 visualize the concepts and the relationships provided by the quality metamodel. Starting point of the quality model are factors, i.e., properties of a software product that are related to quality, which can be refined by more specific factors. If a quality model consists of different factor hierarchies it is possible to define an impact from a factor of one hierarchy to a factor of another hierarchy. For operationalising quality it is necessary to measure factors, whereas these measures can be refined, too.

The basic concepts of the quality model are extended by further concepts as depicted in Figure 3. Both, factors as well as measures, characterize an entities, i.e., parts of the software product or the

environment. While measures just quantify factors with the help of instruments, evaluations provide a facility to use these measures for making qualitative statements about factors.

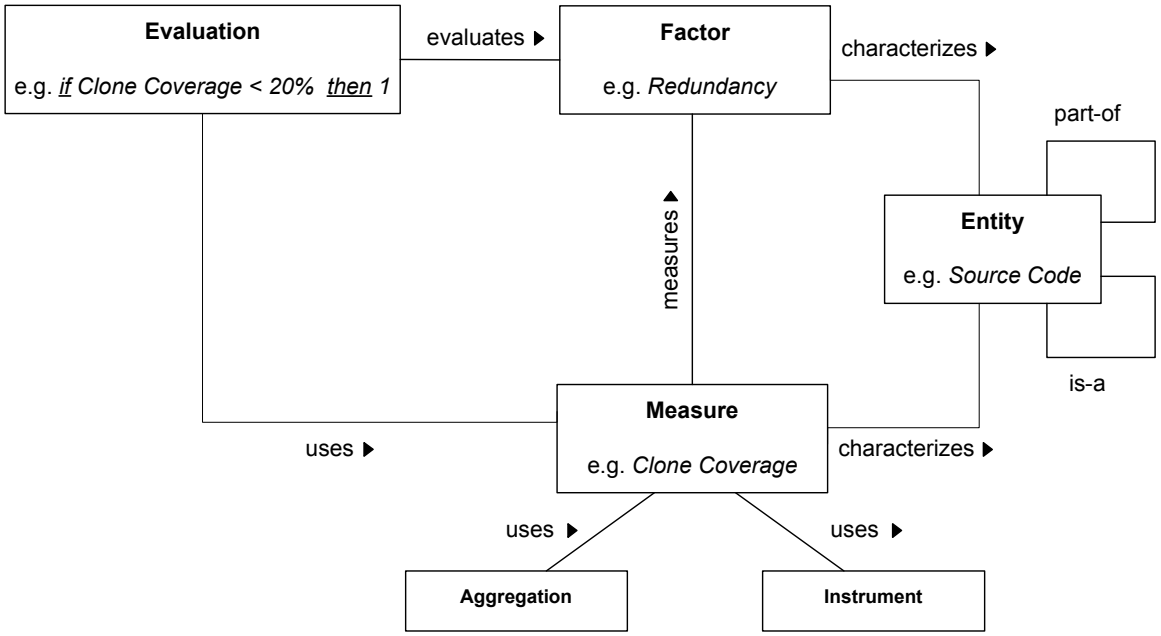

Figure 3: Further concepts of the metamodel on the definition level.

According to the introduced concepts, the definition level of a quality model requires the following constituents:

1. A **factor** constitutes a property of the software product (or part of it) that is related to the product's quality. A factor is always defined in a way that it is possible to determine the degree to which it is present in the product. To indicate that a factor refers to a certain part of the product, the environment of the product or to a resource, the “characterizes” relationship is used to state an entity (see below).
2. A factor can be decomposed into a number of sub-factors by means of a "refines" relationship. Hence, the factors of a quality model are structured by **factor refinements**. The decompose relationship models a refinement of a factor. Since a quality model can focus on different orthogonal views on quality, several hierarchies are possible. Thus, a quality model can accommodate different types of factors like the “ilities” of [ISO 25000] (focus on communication of evaluation results to managers), properties of activities (focus on financial value), technical topics (focus on technical aspects), or requirements (focus on conformity).
3. A factor can influence a factor from another factor hierarchy. The influence is specified by an **impact** relationship. This impact can be either positive or negative. For instance, the "Redundancy at Source Code" has a negative impact on "Maintainability".
4. A **measure** defines how a specific entity is measured and therefore it provides a means to quantify factors that characterize this entity (or a related one). The measurement can be done by using various techniques (by a manual or tool-based **instrument** as well as by an **aggregation** of results of refining measures). Examples are "Lines of Code", "Number of Architecture Violations", or "Clone Coverage".
5. **Entities** are used to model the parts that a software product and its environment consist of. This includes resources that are required during the development or use of the product, like, software developers or other stakeholders. Please note that entities are used to model types of actual entities, i.e. we use the entity “Class” to model the class concept of object orientation – but we do not use entities to model a specific class, as for instance “MyJavaClass.java”. Entities can form a hierarchy, which is defined by “is-a” and “part-of” relationships. For example: a developer is a stakeholder, a class is a part of the source code.
6. An **evaluation** defines criteria to evaluate the extent that a factor is present in a software product. The specification can incorporate references to measures and factors as well as thresholds that constrain their values. An evaluation assesses a factor based on associated measures, based on sub-factors (as part of the corresponding factor hierarchy), or based on factors from other factor

hierarchies that have an impact on the factor being assessed. The evaluation describes how to come up with an evaluation result. The result can be obtained either automatically or by means of expert judgments and is normalized to a value range between 0 and 1. An interpretation scale defines how to present the result.

7. All elements of the quality model can be enriched with additional information. These data are **tags**, i.e., the possibility to mark these elements with keywords, and **sources**, i.e., the possibility to describe the origin of these elements. For keeping the diagram clear, these additional elements are not shown in this overview.

With the illustrated concepts, quality of software products can be defined in a qualitative or in a quantitative manner. For example, one can express that the usability of a user interface is influenced by the consistent usage of fonts and colours or that the maintainability of source code is influenced negatively by reusing source code on a copy & paste base. Hence, a quality model without evaluations can be used as basis of constructive measures of quality assurance, e.g., for developer trainings or the automated generation of guideline documents.

2.2 Application Level

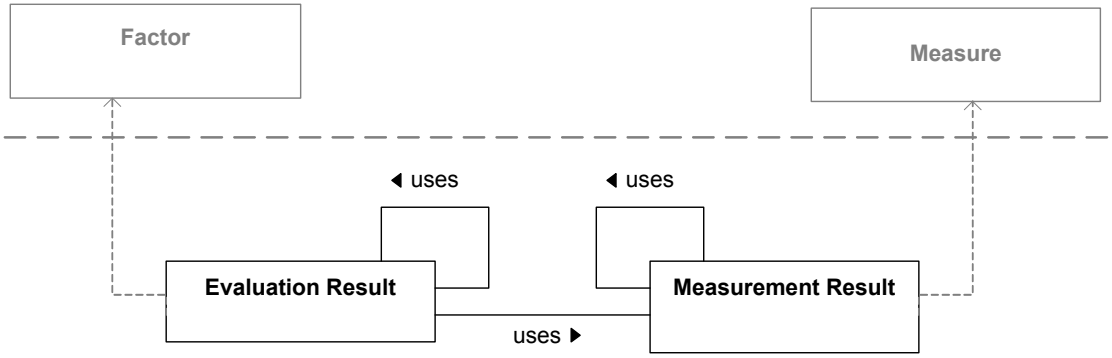

Figure 4: Concepts of the metamodel on the application level.

Figure 4 visualizes the concepts of the application level. The application level provides a means to store measurement and evaluation data for the software product under assessment. For each measure, measurement data is determined as specified by the definition level of the quality model. Based on the measurement results, the evaluation result is determined for each factor. The process of determination is specified by the corresponding evaluation element on the definition level.

3 Detailed Description of the Metamodel Concepts

In this section, we describe the concepts and the relationships defined by the metamodel in more detail by explaining the meaning of each concept.

3.1 Definition Level

3.1.1 Entity

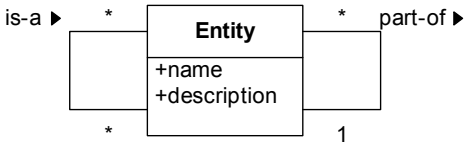

Figure 5: Entity.

An entity is a type of an element which is part of the software product, its environment, or which is a resource required during development, maintenance, or use of a software product. Entities can be structured using two relationships:

1. The “is-a” relationship specifies a generalization/specialization relationship, i.e., the “is-a” relationship is used to specify that an entity is a specialization of another entity. This relation can be used to check the model for completeness regarding factors characterizing the entities.
2. The “part-of” relationship specifies a decomposition of entities into their parts, i.e., the “part-of” relationship is used for decomposition and for navigation in the model. Furthermore, the “part-of” relationship is vital for the tool support.

Identifier: The name of each entity has to be unique.

Example: Example entities that describe product artifacts are “Source Code”, “Method”, “Class”, “Requirements Specification”, “Test Case”, or “Dialog”. Entities that describe resources used during development are “Developer”, “Version Management System”, “Debugger”, or “Review Process”. Examples for relationships are: a "Class" is a part of "Source Code", a "Developer" is a "Stakeholder".

Rationale: Many existing quality models, e.g., the [ISO 9126], do not explicitly model different entities. Hence, properties such as “maintainability” or “changeability” don’t specify the part of the software product or the environment of the product they refer to. However, measuring and correctly evaluating such properties without this information is challenging due to the size, diversity, and complexity of real-world software products. Consequently, we propose to break down products into smaller and more tangible entities.

Remarks: Context-specific data should not be modeled as entities. E.g. programming-language specific entities: Do not use “Java Identifier”, “C Identifier”, “C++ Identifier”, etc.

3.1.2 Factor

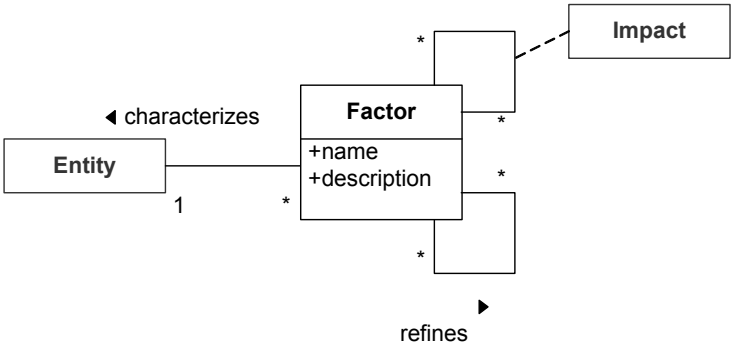

Figure 6: Factor.

Factors are one of the basic building blocks of quality models. A factor constitutes a property of the software product that is related to the products’ quality. A factor is always defined in a way, that it is

possible to talk about the degree to which it is present in the product. The property can be further defined by characterizing an entity. The terms *attribute* and *property* are used interchangeably in the literature. Here, we chose *factor* to clearly distinguish it from the quality attributes defined in ISO 9126 [ISO9126].

Factors can be decomposed into sub-factors by the use of the “refines” relationship. This relationship is used to group related factors as well as to decompose a factor along the line of the “is-a” and “part-of” relationships of its entity. Using the “Impact” relationship can connect different hierarchies of factors formed by the “refines” relationship. The “refines” relationship should result in tree-like structures.

Factors can have a different type, depending on the concrete usage scenario: For instance, a factor can constitute a high-level property of the software product (such as maintainability of software) or a technical property (such as the complexity of the source code). Additionally, it can be used for grouping or classifying. It can also be used for specifying requirements and goals of the software product (making use of the evaluation concept attached to a factor). The type of a factor results from the root factor of the current refinement hierarchy. So the name of a root factor gives a hint to the concept behind the refining factors.

Additionally, the degree of which a software product complies with a factor can give information about the degree of which the product complies with other factors. This relationship is modeled by the “Impact” concept, which provides additional attributes as explained in chapter 3.1.3. Please note that the hierarchy formed by the “Impact” relationship has to be acyclic.

In the remainder of this document, we notate a factor with its name followed by the “@” character prefixing the entity of the factor, as “Name @Entity”. If the factor does not refer to an entity, the factor is simply denoted by its name, as “Name”.

Identifier: The name of a factor in combination with the entity it refers to (Name @Entity) has to be unique.

Example 1: Examples of factors are “Consistency @Identifier” that represents that all identifiers are used consistently throughout a software product or “Existence @Glossary” that represents the existence of a project glossary. The possibility to efficiently run the software is described by “Efficiency @Execute”, whereby the factor “Consistency @Identifier” may have an impact on it. The well-known phenomenon of code cloning is described by “Redundancy @Source Code”, the existence of unnecessary use cases by “Superfluousness @Use Case”. Other examples are “Completeness @Documentation” and “Completeness @Test Case”.

Example 2: Two factors could be “Clarity @UI Element” and “Clarity @Button”. If the “Button” entity is in a “is-a” relationship to the “UI Element” entity, the first factor is in a “decomposes” relationship to the second one. The clarity of buttons is a special form of the clarity of UI elements.

Rationale: The inheritance of entities and the following refinement of factors provide additional means for the structuring of factors and thus enhance clarity. For example, the factor “Clarity @UI Element” is once defined and can then be repeatedly applied to different entities that are subtypes of UI elements, such as, for example, “Clarity @Button” or “Clarity @Scroll Box”. More importantly, the inheritance allows a clean decomposition of quality requirements. This can be illustrated by an example of the quality taxonomy from [Boehm 1978]: “system complexity”. As “system complexity” appears too coarse-grained to be assessed directly, it is desirable to further decompose this element. However, the decomposition is difficult as the decomposition criterion is not clearly defined, i.e. it is not clear what a sub element of system complexity is. A separation of the entity and the factor as in “Complexity @System” allows a cleaner decomposition as entities themselves are not valued and can be broken up in a straightforward manner, e.g. in “Complexity @Sub-System” or “Complexity @Class”.

Remarks: Factors generalize a number of concepts that are available in different specific quality models. Even though these specific concepts are not explicitly modeled in the quality metamodel, they can be expressed as factors. The following are examples for such concepts:

- A factor can be of the type **Product Factor**. These factors are properties of the software product that relate to parts of the software product, such as, “Conciseness @Identifier” or “Correctness @Destructor”.

- A factor can be of the type **Aspect**. Aspects describe high-level factors that are addressed by the quality model. An aspect can be decomposed into sub-aspects resulting in the so-called aspect hierarchy. As a quality model might focus on different orthogonal aspects, several aspect hierarchies are allowed and the metamodel does not prescribe a specific one. Aspects can be related to different types of entities like "Product" or "Activity".
  Examples:
  - **Product Quality Attributes** define one way to decompose the abstract concept software quality. These quality attributes, as given in [ISO 25010] (and the predecessor [ISO 9126].), relate to the quality of the product without explicitly considering its use. These quality attributes are colloquially called -illities, because they contain, for example, reliability or maintainability. In the standard, the top level attributes are refined by so-called quality characteristics.
  - **Quality in Use Attributes** define a way to decompose the abstract concept software quality. These quality attributes are defined in the ISO 25010 as description of the quality in its various forms of usage. Examples are efficiency or effectiveness. To be more precise, we add the activity that is characterised by the attribute as entity to the factor. Activities like maintenance, program comprehension, modification, or testing, which can be decomposed in their respective sub-activities, provide a means to model software development cost structures [Deissenboeck et al. 2007].

  Rationale: Aspects make the view on quality explicit. As different views can be relevant in different situations the proposed metamodel is not limited to a single aspect hierarchy. This allows, e.g., expressing that the factor “Consistency @Identifier” has an impact on maintainability and on the program comprehension activity. While all these statements are obviously related, they address different views of different stakeholders.
- A factor can be of the type **Classification Item**. Classification items are used to group related factors.
  Example:
  - **Technical Issues** as proposed by Siemens/JKU [Plösch et al. 2009] group factors by a technical view. They can be further decomposed in suitable sub-items. Examples are “Memory Issues” or “Procedural Issues”.

  Rational: Classification items allow the grouping of factors by issues areas that are well known to developers and architects.
- A factor can be of the type **Requirement**. Following ISO 9126 [ISO9126], requirements are requested properties of the product. As discussed in WP 2.2 (deliverable 12), we call each factor that is required, a quality requirement. A requirement in the quality model defines constraints to sub-factors.
  Example: An important quality requirement for software is “The code shall be easy to understand”. This could be concretized in the quality model by requiring the factors “Conciseness @Identifier” or “Consistency @Identifier".
  Rationale: Quality requirements are the commonly used basis for specifying and assessing the quality of software products. They are the concept that requirements engineers and developers are familiar with.
- A factor can be of the type **Goal**. Goals are high-level factors that represent the extent a product addresses the needs of a stakeholder. Therefore, goals are root nodes in a factor hierarchy. Furthermore, goals are always referring to a stakeholder by specifying an entity. The factors contributing to a goal are connected to the goal by the “Impact” relationship.
  Example: The goal “Satisfaction @Development Organization” could be defined by requiring a high value of the sub-factor “Efficiency @Maintenance”.
- A factor can be of the type **Stakeholder Satisfaction**. On a high level of abstraction, product quality contributes to the satisfaction of specific stakeholders. This can also be modelled using the stakeholder as an entity. Rationale: This factor hierarchy allows modelling software quality from the view of different stakeholders (for instance "Efficiency @Interaction" or "Correctness @DebugCode").

3.1.3 Impact

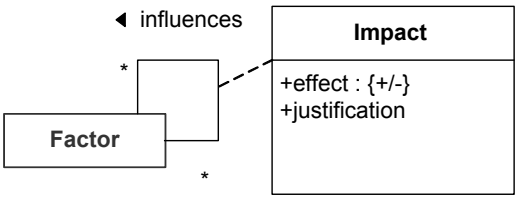

Figure 7: Impact.

The “Impact” concept defines an influence of one factor to another factor that is assigned to a different factor hierarchy. The effect of the influence can be either positive or negative. If the impact has a positive effect, the degree of which the product possesses the target factor is increased if the product possesses the source factor. In contrast, if the impact has a negative effect, the degree of which the target factor is possessed by the product is decreased if the product possesses the source factor. Please note that therefore a positive impact does not mean that the involved factors are required from a quality perspective. For each impact, a detailed justification for the impact needs to be specified. It is crucial to provide such a rationale, since this ensures that the model contains only relevant impacts.

Example: The impact “Redundancy @Source Code → (–)Maintainability” describes that redundant source code is hard to maintain.

Rationale: The “Impact” concept provides a means to relate factors from different refinement hierarchies prior to specify quantitative evaluations. The resulting hierarchy provides helpful information, e.g. it can be used to generate quality guidelines as it defines the essential impacts.

3.1.4 Measure

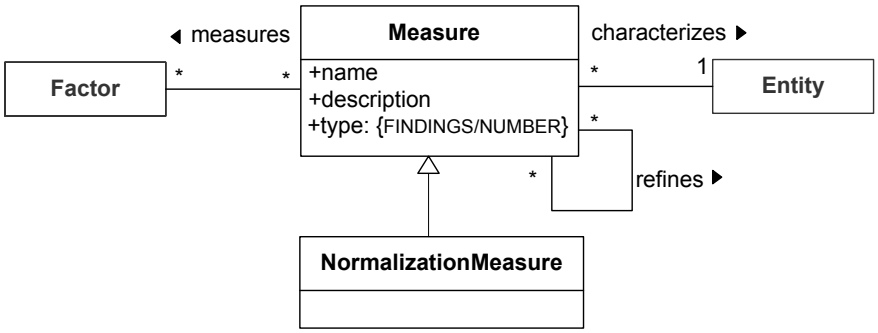

Figure 8: Measure.

The “Measure” concept is used to measure factors and is input for the factor evaluation. A measure defines measurement methods, that is, methods to determine a value for the measure using a certain scale. An entity can be assigned to a measure in order to specify which parts of the product are considered during the measurement. If a measure is refined by other measures than it is recommended to introduce an aggregation to clarify how the refined measure is composed (see the following chapter 3.1.5).

A special measure type is the NormalizationMeasure. These measures are intended to be used only for normalizing other measures when evaluating a factor based on measures.

Identifier: The name of a measure in combination with the assigned entity (if so) has to be unique.

Example: Examples for measures are “Lines of Code (LOC)”, “Clone Coverage”, “Number of Use Cases”, “Test Coverage”, or “Number of Architecture Violations”.

Rationale: To operationalize the quality model, the factors defined by the quality model need to be measured. The metamodel element Measure serves the purpose of defining different measures that can be reused for multiple factors.

3.1.5 Measurement Methods: Aggregations and Instruments

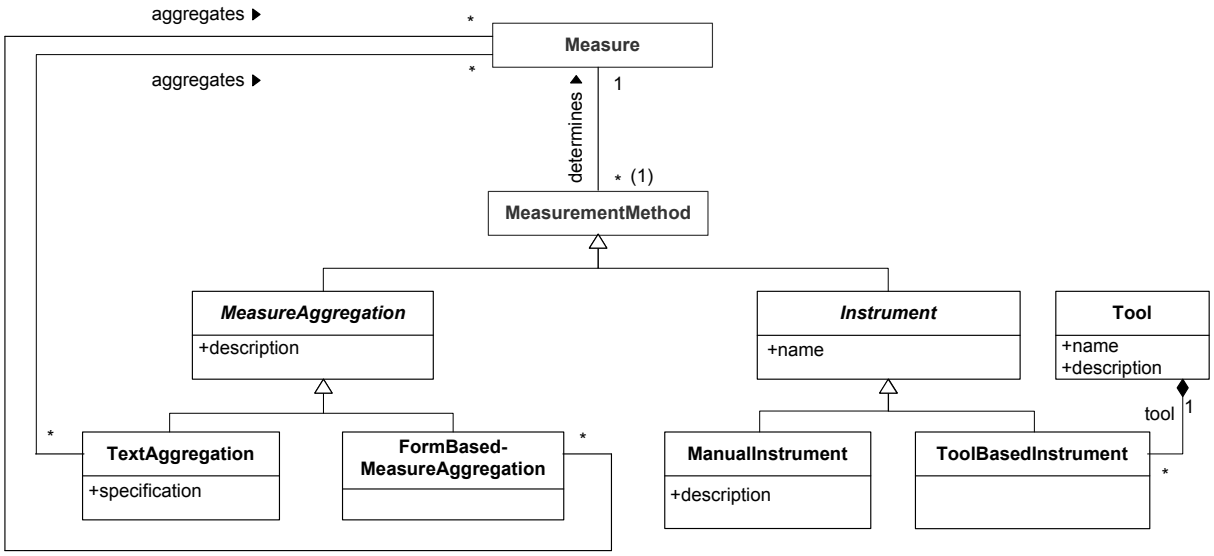

Figure 9: Aggregations and Instruments.

The result of a measure is determined by using two different measurement methods:

1. An **Aggregation** is used to aggregate values of other measures.
2. An **Instrument** is used to determine the value directly using an external tool or a manual assessment.

An aggregation describes how the values of other measures can be aggregated to determine a derived value. The way of aggregating measures can be determined differently; basically with a textual specification or with a restricted aggregation using predefined operations (FormBasedMeasure-Aggregation); see below for currently implemented extensions.

An Instrument describes how the measurement is technically performed. In case of automated measures, the instrument describes the technical connection to a tool; in case of manual measures, the instrument provides a description that explains how the measurement has to be conducted by a person and possibly tells something about the effort.

While on definition level it is possible that various measurement methods determine a single measure, only one method can be used when applying the quality model for measuring (this is expressed by writing a second cardinality for this relation in brackets).

Example: The factor “Validity @Reference” describes that pointer accesses in a software product only access valid memory. A measure for this factor could be “Number of invalid pointer accesses”. To determine a value for this measure we aggregate the results of the application of the PC-lint tool. PC-lint reports such facts with the error codes 413, 794, and 613. Therefore, each result of the particular instruments is assigned to a particular measure, and these three measures are aggregated to determine the value of the measure “Number of invalid pointer accesses”.

Rationale: At some point in the quality model, it is necessary to specify how to determine the values for the measures. The “Aggregation” and the “Instrument” concept provide the means to specify the determination of values.

Extensions for Aggregations in Detail

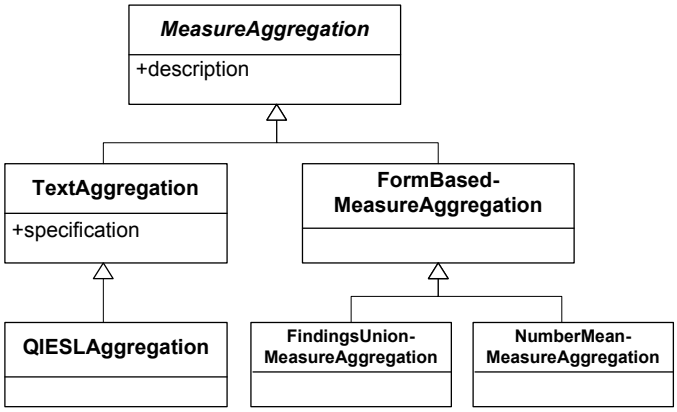

Figure 10: Aggregation in Detail.

This section shows some details for the aggregations that are basically explained above. These additions are currently implemented for the quality model and can be extended if required.

The textual aggregation is extended by a specification with a particular syntax (QIESL) that allows a set of mathematical expressions. With the predefined operations (FormBasedMeasureAggregation) it is possible to make a union for measures that are of type FINDINGS or to calculate the mean for measures that are of type NUMBER.

Example: The measure "Number of invalid pointer accesses" is determined by three measures of the type FINDINGS that are provided by the tool PC-Lint (413, 794, and 613). These three measures can be simple aggregated for the superior measure by using a FindingsUnion-MeasureAggregation:

Measure	Findings
413 - Likely use of null pointer 'Symbol' in [left/right] argument to operator 'String' Reference	17
613 Possible use of null pointer 'Symbol' in [left/right] argument to operator 'String' Reference	134
794 - Conceivable use of null pointer 'Symbol' in [left/right] argument to operator 'String' Reference	89

FindingsUnionMeasureAggregation: Number of invalid pointer accesses	240

3.1.6 Evaluation

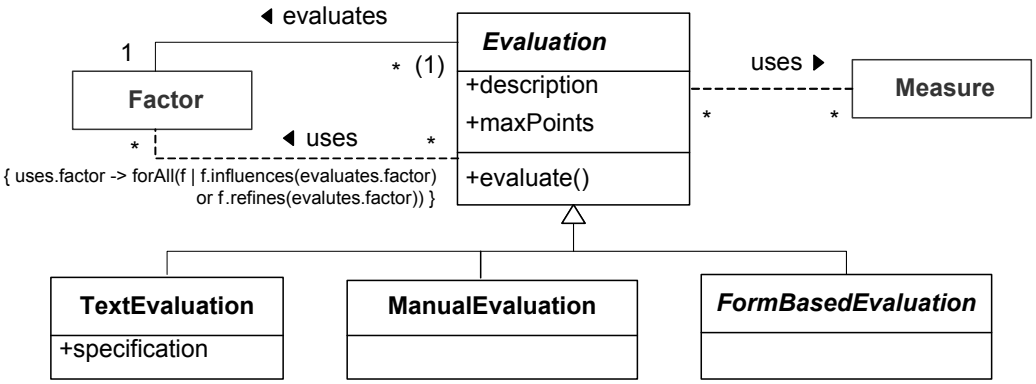

Figure 11: Evaluation.

The "Evaluation" concept provides a mechanism to define how to evaluate a factor. The definition of an evaluation is based on the measures associated to the factor, the sub-factors of the factor, or the factors having an impact on the factor. The definition can be either informal or formal. Basically provided types of evaluations are a manual evaluation, a textual specification or an evaluation with predefined operations (FormBasedEvaluation); see below for currently implemented extensions.

An *evaluate()* method should be implemented by each evaluation and return a result value between 0 and 1. This value will be interpreted for the evaluation by the defined scale; at the moment this scale allows specifying a maximum score.

The quality model specifies one or more possible evaluations, whereby at the time when applying the quality model for a concrete assessment a single evaluation for each factor must be chosen (this cardinality is written in brackets).

There are three possible strategies for evaluating a factor:

1. **Direct**: The measures that are directly associated with the factor are used to evaluate it.
2. **Sub-factors**: The factors that are connected through a “refines” relationship are used.
3. **Impacts**: The factors that are connected through an “Impact” relationship are used.

Example 1: Supposing that scores are used as a scale for the evaluation of the factor “Redundancy @Source Code”, the evaluation using the measure “Clone Coverage @Source Code” could be defined as follows:

Clone Coverage @Source Code (Measure)	Redundancy @Source Code (Factor)	
Measurement Scale (Number)	Evaluation Scale (0..1)	Interpretation Scale (Point)
0 – 5	1.00	100
5 – 15	0.85	85
15 – 25	0.70	70
25 – 35	0.50	50
35 – 50	0.25	25
50 – 100	0.00	0

Please note that this is only an example and that the mapping from measure result to a score will be typically done by using a (linear) function.

Example 2: The evaluation of the factor Maintainability is based on the evaluation results for a factor that refines maintainability (e.g. Modification) and two factors that have an impact on maintainability (Redundancy @Source Code and Completeness @Documentation). The evaluation could be defined as weighted sum of the evaluation scores of these three factors (in this example, the maximum points are 100 for each factor):

Influencing Factor	Weight	Points
Redundancy @Source Code → Maintainability	50%	20
Completeness @Documentation → Maintainability	30%	70
Conformance @Architecture → Modification	20%	40

Weighted Sum:	39

Rationale: The measurement data described by the “Measure” concept needs to be evaluated to fully operationalize the quality model. Many previous approaches to quality modeling and measurement do not clearly distinguish between measurement and evaluation. It has been found that this often leads to a questionable validity, as the approaches tend to compare apples and oranges. Hence, the metamodel proposes a clear separation of the two fundamentally different concepts.

Remarks: At the moment, the only implemented interpretation scale is a linear scale from 0 to MaxPoints. Other scales (e.g. school grades) can be defined if needed.

Extensions for Evaluations in Detail

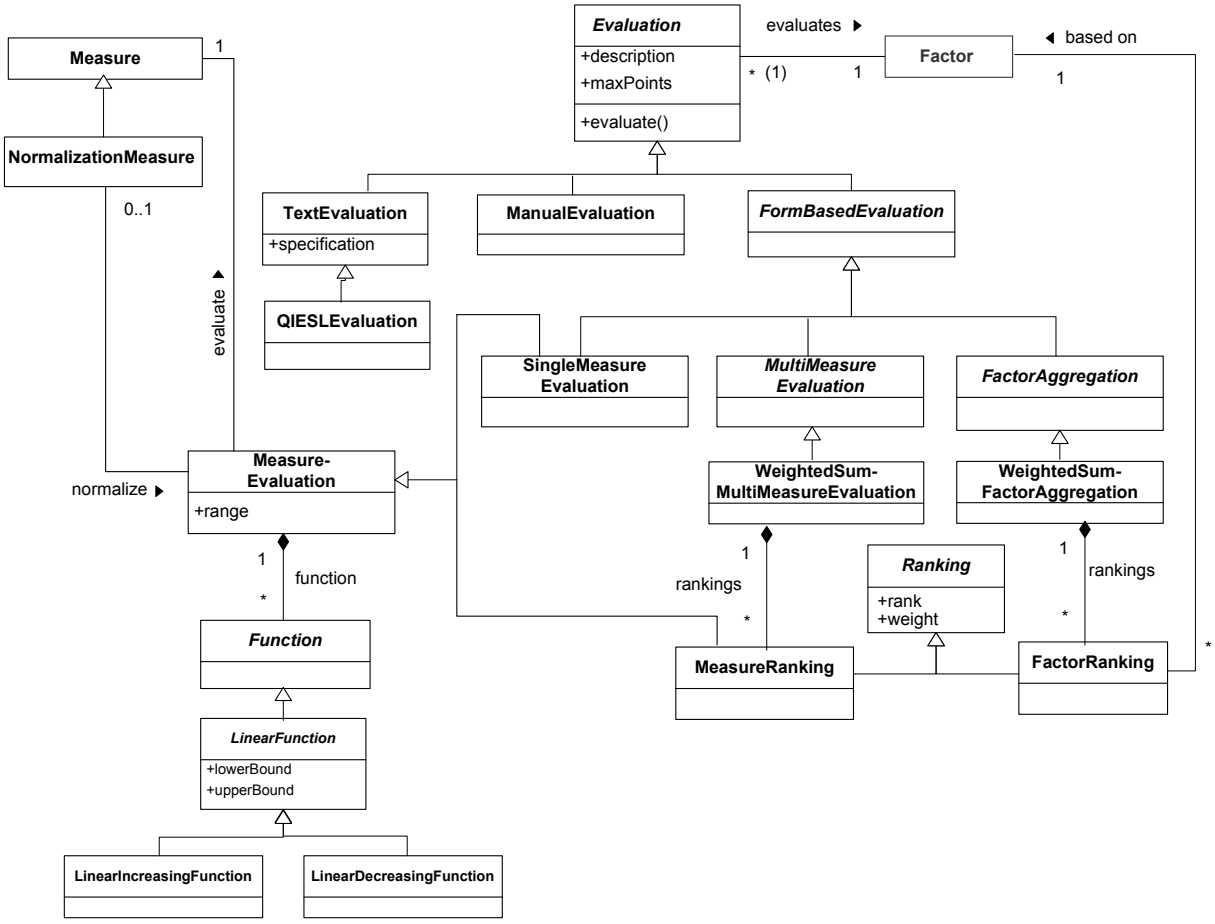

Figure 12: Evaluation in Detail.

This section shows some details for the evaluations that are basically explained above. These additions are currently implemented for the quality model and can be extended if required.

The textual aggregation is extended by a specification with a particular syntax (QIESL) that allows a set of mathematical expressions. With the predefined operations (FormBasedEvaluation) it is possible to evaluate a factor based on a single measure with a linear function (increasing or decreasing), to aggregate the evaluation results of several factors by assigning ranks or weights for the influencing factors or to evaluate multiple measures with a linear function and aggregate them with different weights for one factor. When using a linger function for a measure, there is the possibility to specify a normalization measure or to define a range (e.g. on method or class level) to identify the ratio between the affected code and the total source code.

Example 1: According to example 1 in the section before we evaluate “Redundancy @Source Code” with the measure “Clone Coverage @Source Code”. Using a SingleMeasureEvaluation we define a LinearDecreasingFunction with lowerBound=0 (%) and upperBound=100 (%). For clone coverage of 15% we get the evaluation result 0.85 (which will be interpreted as 85 points if MaxPoints are 100). If we have no measure that detects clone coverage directly, we could use a measure "Number of Cloned Lines" and use "Lines of Code" as normalization measure for our evaluation.

Example 2: According to example 1 in the section before we evaluate the factor Maintainability by three other factors with a WeighedSumFactorAggregation. The influence of these factors is determined by their weights (0.5, 0.3, and 0.2) and results to 0.39 (i.e., 39 points if MaxPoints is 100) if the evaluation results for the three factors are 0.2, 0.7, and 0.4.

3.1.7 Additional Information: Tag and Source

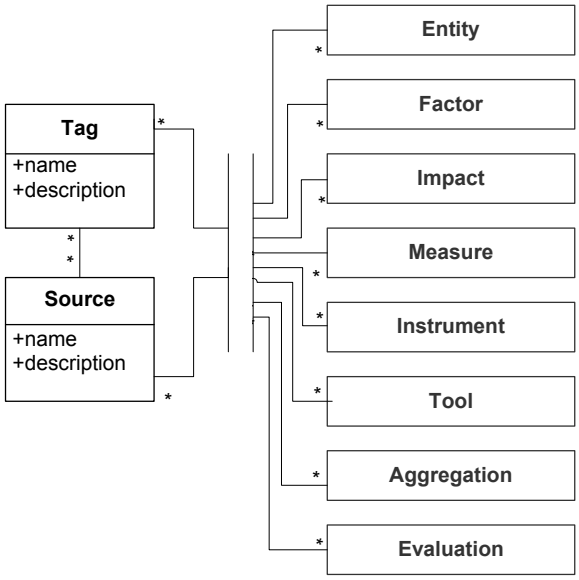

Figure 13: Tag and Source.

Tags allow assigning keywords to model elements. A tag has a unique name that should be short and concise to express the idea underlying the tag. Optionally it is possible to add a description for a tag. Tags can be used when applying the quality model for a project to flag the progress of some process (e.g. the current assessment process) or to show some other unforeseen information.

Sources have a unique name and an optional description and give hints about the origins of the particular model elements.

Example: Tags that are used to flag the progress of the process can be for instance "Evaluation in work" or "Evaluation completed". Possible sources are for instance "ISO 25010" or "IEEE Std 1219-1998".

Rationale: Tags can help to filter elements by any criteria. Sources can cite the origin of the particular model element and reference to additional background information.

3.2 Application Level

3.2.1 Measurement and Evaluation Results

When measures and evaluations are applied to the software product, concepts are needed to store their results. The following figure illustrates the results of measures and evaluations:

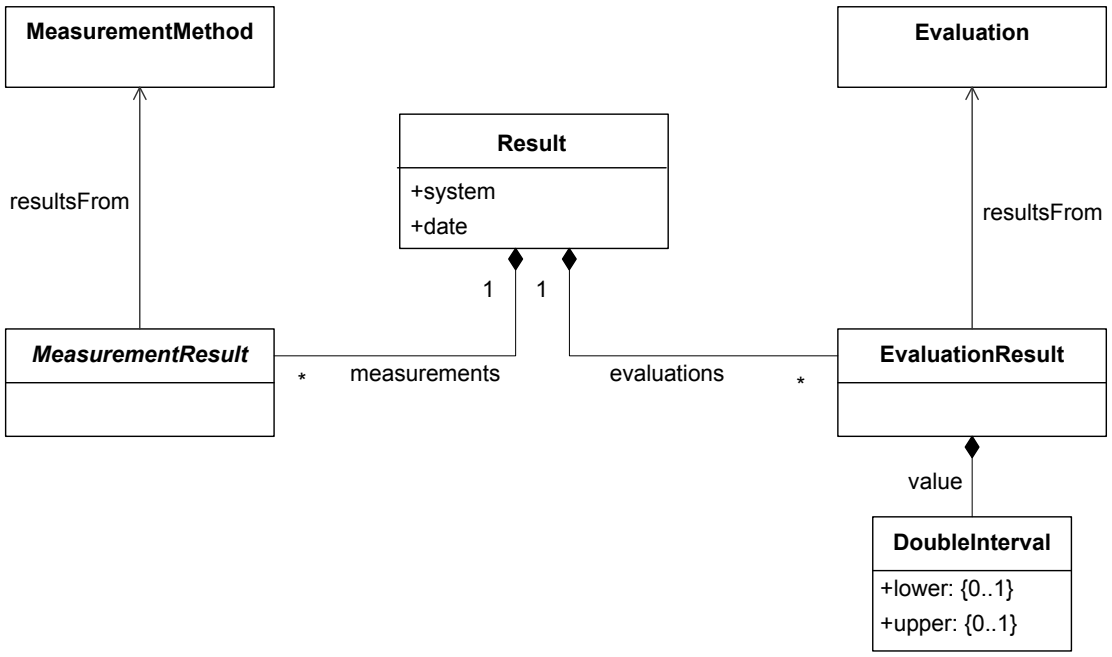

Figure 14: Measurement and Evaluation Results.

Measurement Result: A measurement result contains the result of a measurement performed on a software product. The measurement is performed by using the corresponding measurement method as defined on the definition level. If the measure uses an aggregation of other measures, then the measurement result is based on the measurement results of the other measures and is obtained by applying the specification of the aggregation. The type of the measurement result value depends on the type of the measure and therefore on the instrument that performs the current measure (number or findings, see chapter 3.2.2).

Example: The measure “Clone Coverage” determines the probability that a random code line is part of a clone. When it is applied to the source code of the software product, the result is a measurement result with the unit percentage, e.g. Clone Coverage = 0.2 (20%).

Rationale: The existence of this metamodel element is a direct consequence of the separation between definition and application level.

Evaluation Result: An evaluation result contains the result of the evaluation of a factor. The evaluation is performed as defined by the specification of the corresponding evaluation on the definition level. An evaluation of a factor can be based on the measures associated to the factor as well as on the evaluations of the factors which decompose the factor or which have an impact on the factor. Likewise, the evaluation result is based on the measurement results collected for all the required measures as well as on the evaluation results collected for all the required factors. The result value of the evaluation has a range between 0 and 1 and can be interpreted by the evaluation's scale (MaxPoints). The value is stored as an interval with a lower and an upper value to allow some uncertainty because of incomplete data. Furthermore, there are currently additional evaluation results implemented for the extended evaluations (details about the extended evaluations can be found in the extension section of chapter 3.1.6 and details about the according evaluation results can be found below).

Example: Based on the measure “Clone Coverage“, we can evaluate the factor “Redundancy of Source Code”. The specification of the evaluation distributes the percentages obtained from the measure over maximum reachable points. For a “Clone Coverage” of 20%, the evaluation result would be a for instance 0.7, i.e. 70 points on the interpretation scale (out of 100 MaxPoints).

Rationale: The existence of this metamodel element is required to store the evaluation result for a factor - either based on measures or based on derived factors.

Extensions of Evaluation Results in Detail

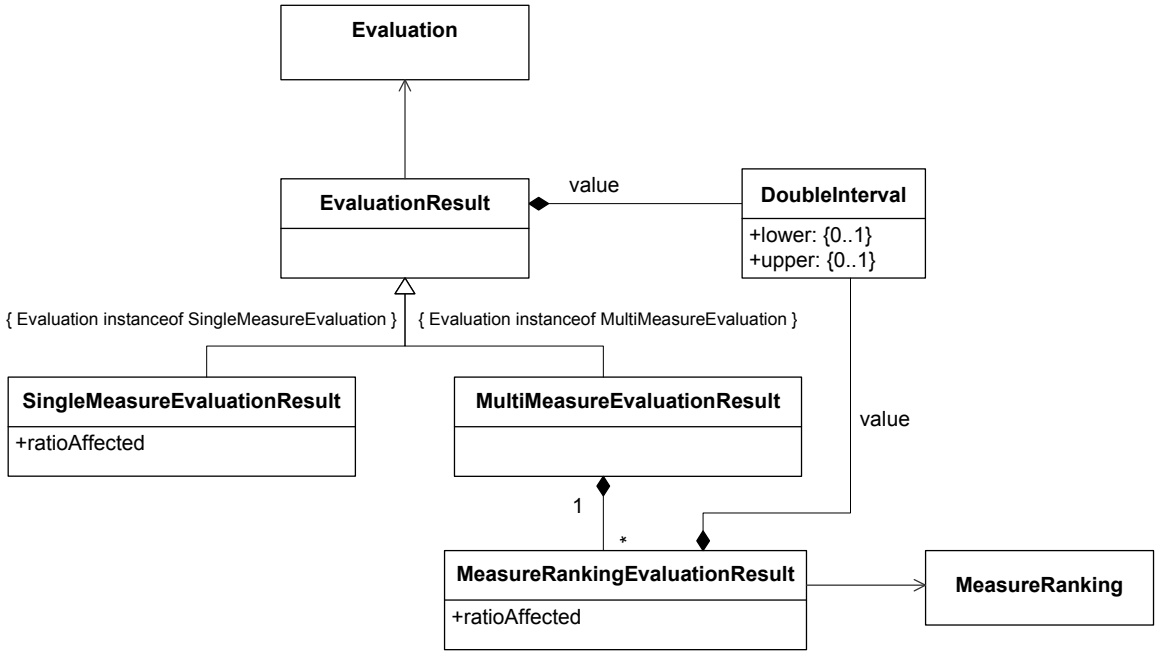

Figure 15: Evaluation Result in Detail.

This section shows some details for the evaluations that are basically explained before. These additions are currently implemented for the quality model and necessary for the extension of the evaluation as shown in chapter 3.1.6. All result values are stored as intervals with a lower and an upper value to allow some uncertainty because of incomplete data.

For a SingleMeasureEvaluation the result is stored as SingleMeasureEvaluationResult. The attribute ratioAffected stores the ratio between affected code by this measure and the total source code. This information is only available if there is a range specified at the corresponding measure evaluation.

For a MultiMeasureEvaluation the result is stored as MultiMeasureEvaluationResult. Each intermediate result one measure is stored as MeasureRankingEvaluationResult and connected with the particular MeasureRanking. Similar to the single measure result there is a ratioAffected attribute to store the ratio between affected code and total source code.

Example: These elements store evaluation results for factors that are retrieved by evaluating measures. According to the examples in chapter 3.1.6, we store the evaluation result of 0.85 for the measure "Clone Coverage @Source Code" as SingleMeasureEvaluation-Result. As this measure has no range, the ratioAffected attribute is not set. The evaluation result for Maintainability (0.39) is not stored, because it can be recalculated by using the evaluation results of the three influencing factors whose evaluation results can be either re-calculated, too, or are stored as results of measure evaluations.

3.2.2 Values of Measurement

The types of measurement results depend on the type of the measure. The following figure illustrates the basic types which are needed for storing measurement results:

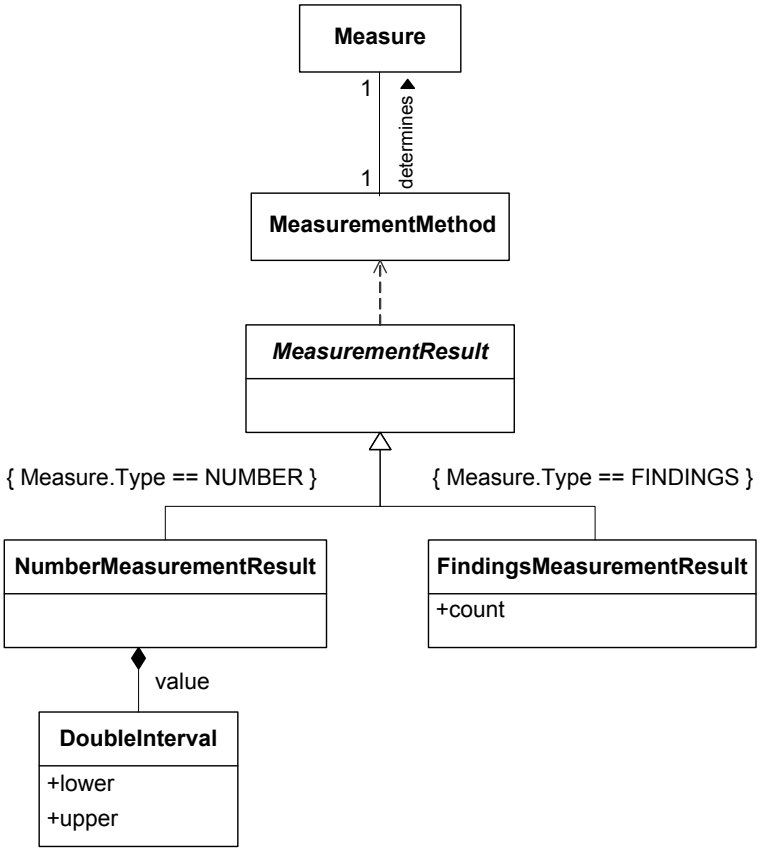

Figure 16: Types of measurement results.

NumberMeasurementResult: The measured value is a number with any meaning, depending on the description of the measure. The value is stored as an interval with a lower and an upper value to allow some uncertainty because of incomplete data.

FindingsMeasurementResult: A finding represents the violation of a rule in the software product. For the evaluation we are interested in the number of findings ("count") for one measure. At the moment no additional information (e.g. location of a finding) is required.

4 Modularization Concept

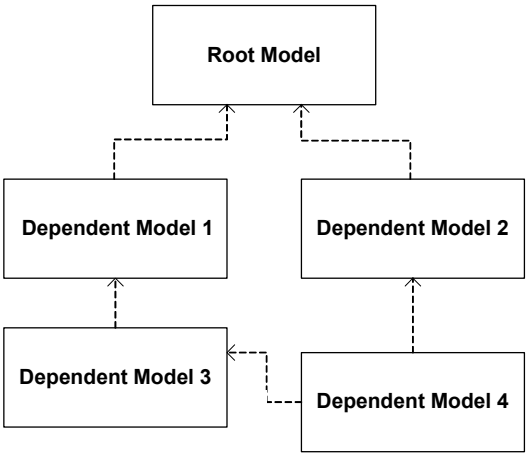

Figure 17: The modularization concept.

A quality model can be separated into modules (files). As you can see in Figure 17 the dependencies between models are hierarchical, i.e., there is one root model and depending models can reference to one or more superior models. Contrary, superior models do not reference to any subordinate (depending) model. Depending model can, for instance, extend the root quality model by refining factors with new specific factors, by measuring factors with new measures, by adding new evaluations for factors or by introducing new instruments and tools for existing measures.

Every quality model can introduce new elements of all available types which can be used in the current model. Some types can use elements of superior models (and vice-versa new elements can be used again in depending models). The following references from a depending model to a superior one are possible:

- A new factor can refine a factor in a superior model.
- A new factor can have an impact on a factor in a superior model.
- A new evaluation can evaluate a factor in a superior model.
- A new measure can measure a factor in a superior model.
- A new measure can refine a measure in a superior model.
- A new aggregation can be defined for a measure in a superior model.
- A new instrument can be assigned to a measure in a superior model.

A reference from a dependent model to a superior model can overwrite the corresponding element there, i.e., only the most specific element is used. For instance of there exists an evaluation for an factor in the root model and a depending model provides an evaluation for the same factor, then the original evaluation will be ignored and only the evaluation of the depending model is valid (as long as the depending model is in use).

5 Example Model: Cloning of Source Code

To demonstrate the usage of the metamodel, a small example model is presented. In this example, a small model that defines quality in terms of cloning of source code is shown. Figure 18 illustrates the example model as a diagram. Each rectangle of the diagram denotes an instance of a metamodel concept (namely the concept mentioned after the colon). Each line in the diagram denotes a relationship as defined by the metamodel (namely the relationship mentioned as label). The example model for cloning of source code makes use of most concepts and relationships defined by the metamodel.

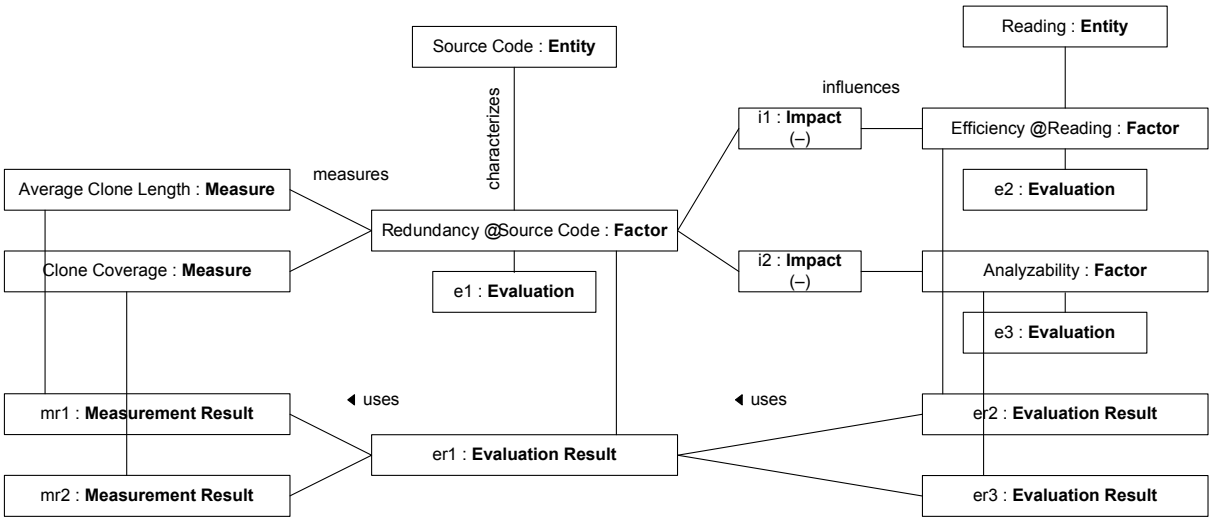

Figure 18: Example model visualized as a diagram

As a starting point, we define the following model elements:

1. Comprehending existing code is an essential activity in software maintenance. To express this, we use the factor “Efficiency @Reading”.
2. We know that code cloning has a negative impact on the reading activity and, hence, introduce an impact with a negative effect on the factor.
3. Code cloning manifests itself in the source code. Therefore, we introduce the entity “Source Code”.
4. Code cloning is a special form of redundancy. We express this by defining a factor “Redundancy @Source Code” that refers to the entity “Source Code”.
5. This factor is associated with the earlier defined impact to express that redundant source code negatively affects the effectiveness of reading source code.
6. Next to the activities, our model should also include the so called “ilities”. In a bottom-up manner, we consider which “ilities” are influenced by the factor “Redundancy @Source Code”. To express, that code cloning negatively influences “Analyzability”, we introduce a second impact that associates the factor with the “ility”.

For measuring and evaluating, we define the following quality model elements:

1. We plan to quantify the factor “Redundancy @Source Code” by means of two measures. The first measure determines the average length of clones in lines of code. Therefore, we associate a new measure called “Average Clone Length” to the factor. The second measure determines the probability that a random code line is part of a clone. Therefore, we associate a new measure called “Clone Coverage” to the factor.
2. Based on these measures, we can now define an evaluation for the factor. The higher the *Clone Coverage*, the worse the verdict of the evaluation. A higher average length of code clones further adulterates the evaluation. We create an evaluation for the factor, and refer to both measures through the use relationship.

3. For the two remaining factors, we can now define an evaluation which has access to the factor evaluations of all incoming impacts. There is only one impact on the factor “Efficiency @Reading” that can be used to evaluate the factor. Consequently, the evaluation has to interpret the verdict for the impact in terms of the activity. The procedure is similar for the factor “Analyzability”.

When we have defined the evaluation, we can execute it on a concrete software product, e.g., the editor for quality models developed as part of Quamoco:

1. We evaluate the *QM Editor* by looking at its source code.
2. We apply the measures on the source code which results in measurement results. For example, we measure an average clone length of 10, and clone coverage of 20%.
3. Based on the measurement results, we apply the factor evaluation which leads to an evaluation result. This evaluation aggregates the corresponding measurement results into a grade as defined by the evaluation. To determine the evaluation result we take both measurement results into account.
4. Based on the evaluation result “e1”, we apply the evaluation for the factors “Efficiency @Reading” and “Analyzability” which results in further evaluation results.

6 Example Model: Accessibility of User Interfaces

The following small example demonstrates how to use the metamodel for modeling quality as compliance to quality requirements. Accordingly, the example model defines quality in terms of the accessibility of a user interface that is used by people with limited color vision. Using the same notation as in Section 4, Figure 19 illustrates the model as a diagram.

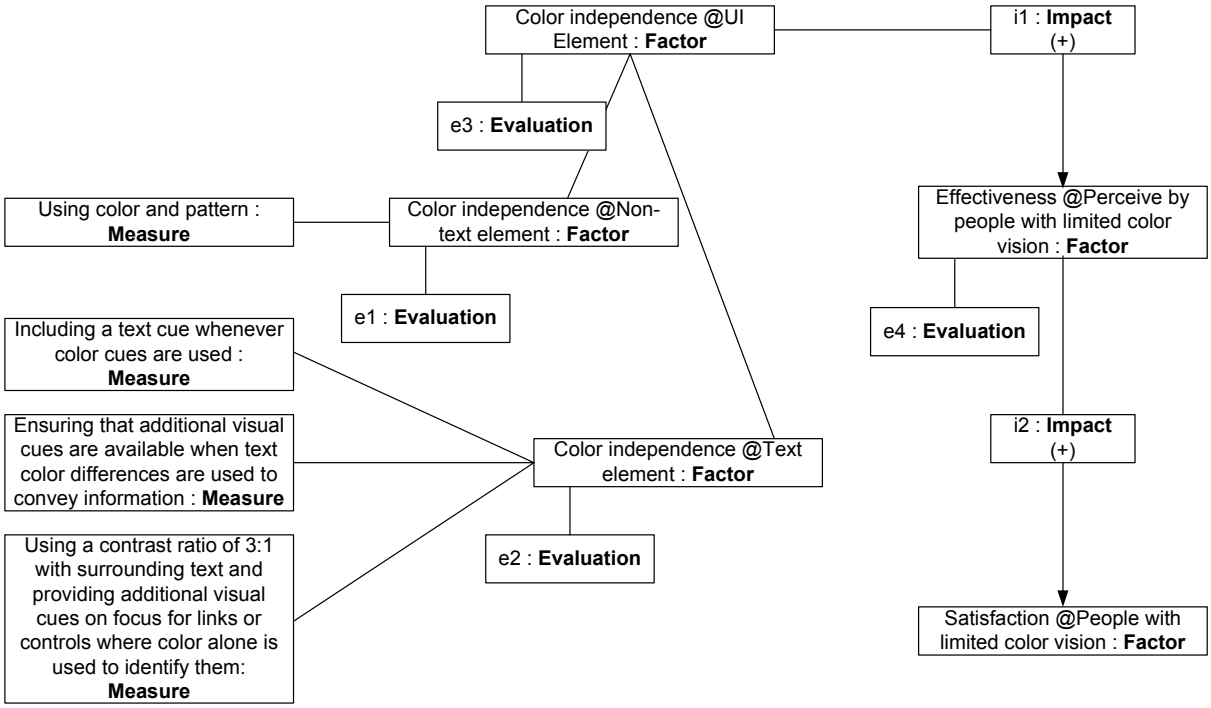

Figure 19: Accessibility example model visualized as a diagram

First we create the core elements of our quality model and define the following factors:

1. For this example, we assume that a high-level accessibility goal demands that the needs of people with limited color vision are satisfied by the software product. We model this goal by the factor “Satisfaction” that relates to the stakeholder “People with limited color vision”. (A stakeholder, in this example, is modeled as an entity that feature an “is-a” relation to the “Stakeholder” entity.)
2. To fulfill the goal, the factor defines that several use cases need to be accomplishable in an effective manner, as, for example, the use case “Perceive”. (To indicate the actor, the name of the use case is amended by “people with limited color vision”.) For the precise definition of “effective manner”, the factor “Effectiveness” that relates to the use case “Perceive by people with limited color vision” is modeled. (A use case is modeled as an entity that feature an “is-a” relation to the

"Use Case" entity.) The relationship between the use case-based factor and the stakeholder-based factor is captured by an impact that is defined from the first to the latter. The impact is a positive one, since the more effectively the software product can be perceived by people with limited color vision, the more satisfied they are with the product.

3. A software product is effectively perceived by people with limited color vision, if the user interface is independent of colors. For the precise definition of color independence, the factor "Color independence" that relates to the entity "UI Element" is modeled. (To indicate that an "UI Element" is a part of the product, it is linked to the entity "Product Part" with an "is-a" relation.) The relationship between the product-based factor and the use case-based factor is captured by an impact that is defined from the first to the latter. The impact is again a positive one, since the more color independent a UI element is, the more effectively the software product can be perceived by people with limited color vision.
4. We refine the factor "Color independence @UI Element" by looking at specializations of UI elements. For example, an element of the UI can be either a non-text element or a text element. This knowledge is modeled by two entities that feature an "is-a" relationship to the "UI Element" entity. Consequently, we create the refined factors "Color independence @Non-text element" and "Color independence @Text element".

Then we extend the model by the ability to measure and evaluate it by defining the following elements:

1. The factors "Color independence @Non-text element" and "Color independence @Text element" are sufficiently concrete. Hence, we determine them directly by measures. To measure the factor "Color independence @Non-text element", we define the measure "Using color and pattern". This manual measure requires a reviewer to check whether non-text elements that encode information in colors provide appropriate patterns to display this information without colors. In the same way, we can also define measures for the other factor "Color independence @Text element".
2. Based on the measures, the factor "Color independence @Non-text element" can now be evaluated. Therefore, we define an evaluation for the factor that returns findings for non-text elements that are not color independent. For the factor "Color independence @Text element", a similar evaluation can be defined based on its measures.
3. Based on these evaluations, we can define an evaluation of the factor "Color independence @UI Element". This evaluation unites the findings produced by the evaluations and sets them in relation to all UI elements to obtain a grade. The percentage of non-conforming UI elements is linearly distributed over school grades.
4. Based on the evaluation of the product-based factor, we can define an evaluation for the use case-based factor "Effectiveness @Perceive by people with limited color vision". Since there might be also other factors influencing or refining this factor, we specify the evaluation as a median over all the grades that are returned by their evaluations.
5. In the same way, we could also define an evaluation for the stakeholder-based factor "Satisfaction @People with limited color vision".

7 References

[ISO 25010] ISO/IEC 25010: Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models, 2011

[ISO 9126] DIN ISO 9126: Informationstechnik – Beurteilen von software projects, Qualitätsmerkmale and Leitfaden zu deren Verwendung, 1991

[Plösch et al. 2009] Plösch R., Gruber H., Körner C., Pomberger G., Schiffer S.: A Proposal for a Quality Model Based on a Technical Topic Classification, 2. Workshop zur Software-Qualitätsmodellierung und –bewertung (SQMB '09), March 3, 2009, Kaiserslautern, Germany, 2009

[Deissenboeck et al. 2007] Deissenboeck, F., Wagner, S., Pizka, M., Teuchert, S., Girard, J.-F.: An Activity-Based Quality Model for Maintainability. In Proc. 23rd International Conference on Software Maintenance (ICSM '07). IEEE Computer Society, 2007.

[Kitchenham et al. 2001] Kitchenham, B., Hughes, R., Linkman, S.: Modeling Software Measurement Data. IEEE Transactions on Software Engineering, vol. 27, no. 9, pp. 788-804, Sept. 2001

[Boehm 1978] Boehm, B.: Characteristics of Software Quality, TRW Series of Software Technology. Elsevier Science Ltd. June 1978

8 Appendix

8.1 Simplified Meta Model as UML Class Diagram