An Activity-Based Quality Model for Maintainability

F. Deissenboeck, S. Wagner, M. Pizka
Institut für Informatik
Technische Universität München
Boltzmannstr. 3
85748 Garching b. München, Germany
{deissenb,wagnerst,pizka}@in.tum.de

S. Teuchert, J.-F. Girard
MAN Nutzfahrzeuge AG
Elektronik Regelungs- und Steuerungssysteme (TSE)
Dachauer Strasse 667
80995 München, Germany
{stefan.teuchert,jean-francois.girard}@de.man-mn.com

Abstract

Maintainability is a key quality attribute of successful software systems. However, its management in practice is still problematic. Currently, there is no comprehensive basis for assessing and improving the maintainability of software systems. Quality models have been proposed to solve this problem. However, existing approaches do not explicitly take into account the maintenance activities, that largely determine software maintenance effort. This paper proposes a 2-dimensional model of maintainability that explicitly associates system properties with the activities carried out during maintenance. The separation of activities and properties facilitates the identification of sound quality criteria and allows to reason about their interdependencies. This transforms the quality model into a structured and comprehensive quality knowledge base that is usable in real project environments. For example, review guidelines can be generated from it. The model is based on an explicit quality metamodel that supports its systematic construction and fosters preciseness as well as completeness. An industrial case study demonstrates the applicability of the model for the evaluation of the maintainability of Matlab Simulink models that are frequently used in model-based development of embedded systems.

1. Introduction

Virtually any software dependent organization has a vital interest in reducing its spending for software maintenance activities. In addition to financial savings, for many organizations, the time needed to complete a software maintenance task largely determines their ability to adapt their business processes to changing market situations or to implement innovative products and services. That is to say that with the present yet increasing dependency on large scale software systems, the ability to change existing software in a timely and economical manner becomes critical for numerous enterprises of diverse branches. The term most frequently associated with this ability is *maintainability*. But what is maintainability? An often cited definition of maintainability is

> *The effort needed to make specified modifications to a component implementation*[1],

This nicely illustrates that the desire for *high maintainability* is really a desire for *low maintenance efforts*. However, current approaches to assess and improve maintainability fail to explicitly take into account the cost factor that largely determines software maintenance efforts: the *activities* performed on the system or more precisely, the associated personnel costs. Considering the diverse nature of activities, such as "problem understanding" and "testing", it becomes evident, that the criteria that actually influence the maintenance effort are numerous and diverse. Psychological effects, such as the *broken window* [26] deserve just as much attention as organizational issues (e. g. personnel turnover) or properties of the code. Any of these aspects may have a significant and vastly independent impact on the future maintenance effort.

We regard the omission of activities as a serious flaw not only due to the activities' major importance for the overall maintenance effort but also because the activities provide a natural criterion for the decomposition of maintainability that many existing approaches lack.

Problem Although maintainability is a key quality attribute of large software systems, existing approaches to model maintainability have not created a common understanding of the factors influencing maintainability and their interrelations. Hence, no comprehensive basis for assessing

---

[1] SEI Open Systems Glossary (`http://www.sei.cmu.edu/opensystems/glossary.html`)

and improving the maintainability of large software systems has been established so far.

Typically, existing models exhibit at least one of the following problems: First, they do not decompose the attributes and criteria to a level that is suitable for an actual assessment. Second, these models tend to omit the rationale behind the required properties of the system. Third, existing models often use heterogeneous decomposition dimensions, e. g. the required criteria mix properties of the system with properties of the activities carried out on the system.

The first problem constrains the use of these models as the basis for analyses. The second one makes it difficult to describe impacts precisely and therefore to convince developers to use it. The third problem leads to inconsistent models and hampers the revelation of omissions and inconsistencies in these models.

Contribution This paper proposes a 2-dimensional model of maintainability that explicitly associates system properties with the activities carried out during maintenance and thereby facilitates a structured decomposition of maintainability. The separation of activities and properties facilitates the identification of sound quality criteria and allows to reason about their interdependencies. As the activities are the main cost factor in software maintenance, we consider this separation a first step towards the ultimate goal of a truly economically justified practice of maintainability engineering. The model is based on an explicit quality metamodel that supports a systematic construction of the maintainability model and fosters preciseness as well as completeness.

Next to the ability to explicitly describe the impact of system properties on the maintenance activities, several additional benefits can be derived by using the model:

1. The model provides a central storage of quality definition, comparable to a knowledge base, that serves as a basis for the automatic generation of guideline documents for specific maintenance tasks as well as for the analysis of system artifacts.

2. The model allows to reveal omissions and contradictions in current models and guidelines.

We demonstrate the applicability of the 2-dimensional model in a case study undertaken with MAN Nutzfahrzeuge, a German supplier of commercial vehicles and transport systems. Here we created a comprehensive model of the maintainability of Matlab Simulink models that are frequently used in model-based development of embedded systems. The study lead to the inclusion of the model into the MAN standard development process.

2. Related Work

To be read conveniently the related work is categorized as guidelines-based approaches, metrics-based approaches, quality models and process-based approaches.

Guidelines A commonly applied practice are guidelines that state what developers should do and what they should not do in order to improve the quality of software artifacts. Such guidelines are usually composed by the software-developing companies itself or by tool providers, e. g. the Java Coding Conventions provided by Sun Microsystems [23].

Unfortunately, such guidelines typically do not achieve the desired effect as developers often read them once, tuck them away at the bottom of a drawer and follow them in a sporadic manner only. According to our experience [3], this is often due to the fact that guidelines fail to motivate the required practices or provide very generic explanations, e. g. “Respecting the guideline ensures readable models” in [17]. Justification could be provided by explaining how conformance/non-conformance to guidelines effects maintenance activities and thereby maintenance effort.

In addition to this, guidelines are often not followed simply because it is not checked if they are followed or not. This is all the more unfortunate as for some guidelines rules compliance could be assessed automatically.

Metrics-based Approaches Several groups proposed metrics-based methods to measure attributes of software systems which are believed to affect maintenance, e. g. [1,4]. Typically, these methods use a set of well-known metrics like *lines of code*, Halstead volume [9], or McCabe’s Cyclomatic Complexity [18] and combine them into a single value, called *maintainability index* by means of statistically determined weights.

Although such indices may indeed often expose a correlation with subjective impressions and economic facts of a software system, they still suffer from serious shortcomings. First, they do not explain in which way system properties influence the maintenance activities and thereby the overall maintenance efforts. This makes it hard to convey their findings to the developers.

Second, they focus on properties which can be measured automatically by analyzing source code and thereby limit themselves to syntactic aspects. Unfortunately, many essential quality issues, such as the usage of appropriate data structures and meaningful documentation, are semantic in nature and can inherently not be analyzed automatically.

Because of this, most known metrics, such as the Cyclomatic Complexity, are neither sufficient nor necessary to indicate a quality defect. Therefore, individual metrics or simple indices provide only a poor basis for effective quality assessments.

Quality Modeling A promising approach developed for software quality in general are *quality models* which aim at describing complex quality criteria by breaking them down into more manageable sub-criteria. Such models are designed in a tree-like fashion with abstract quality attributes like *maintainability* or *reliability* at the top and more concrete ones like *analyzability* or *changeability* on lower levels. The leaf factors are ideally detailed enough to be assessed with software metrics. This method is frequently called the decompositional or *Factor-Criteria-Metric* (FCM) approach and was first used by McCall [19] and Boehm [2].

Unfortunately, these and more recent approaches like [7, 11, 15, 20] have failed to establish a broadly acceptable basis for quality assessments so far. We believe this is due to the lack of a clearly defined decomposition criterion that leads to a "somewhat arbitrary selection of characteristics and sub-characteristics" [12, 13]. Moreover, we see their fixed number of model levels as a problem. For example, FCM's 3 level structure is inadequate. High level goals like *maintainability* cannot be broken down into assessable properties in only two steps.

Similar to other approaches, quality models do usually not explicitly model the maintenance activities. Hence, they are not directly capable of explaining how system properties influence the maintenance effort.

Processes and Process Models Organizational issues are typically covered by process-based approaches to software quality like the ISO 9000 standards or CMM [21]. Unfortunately, there is the widely disputed misconception, that good processes automatically guarantee high quality products [13]. Of course, processes are of high importance and they determine reproducibility of the development process. However, the quality of the outcome still strongly depends on the actual criteria, skills, and tools used during development.

Discussion There is an abundance of further highly valuable work on software quality in general and maintainability in particular that we do not explicitly mention here, as it is either out-of-scope or does not fundamentally differ from the work already mentioned. Overall, this is and has been a very active field of research which definitely does not need to be reinvented by itself.

However, we argue that existing approaches to assess and improve software maintainability generally suffer from one or more of the following shortcomings:

1. *Assessability.* Most quality models contain a number of criteria that are too coarse-grained to be assessed directly.

2. *Justification.* Additionally, most existing quality models fail to give a detailed account of the impact that specific criteria (or metrics) have on software maintenance.

3. *Homogeneity.* Due to the lack of a consistent criterion of decomposition most existing models exhibit inhomogeneous sets of quality criteria.

4. *Operationalization.* Most times, quality models are expressed in prose and graphics only. They accompany the development process in the form of documents but are not made an integral artifact that is tightly coupled with the quality assurance activities.

3. Maintainability Model

To address the problems of quality models described in the previous section we developed a novel two-dimensional quality model. The initial version of the model was developed in the context of a commercial project in the field of telecommunication [3]. As the analyzed system was large (3.5 MLOC[2] C++, COBOL, Java), 15 years old and under active maintenance with 150 change requests per year it was well suited for an application of our quality model.

In contrast to other quality models that are expressed in terms of prose and graphics only, our maintainability model is truly integrated in the software development as basis of all quality assurance activities. As Fig. 1 shows, the model can be seen as project- or company-wide quality knowledge bases that centrally stores the definition of quality in a given context. Of course, an experienced quality engineer is still needed for designing the quality models and enforcing them with manual review activities. However, he can rely on a single definition of quality and is supported by the automatic generation of guidelines. Moreover, quality assessment tools like static analyzers that automatically assess artifacts 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 in the model. We refer to this approach as *model-based quality controlling*.

The following sections explain the basic concepts of our model and discuss the differences to classical hierarchical models.

---

[2] million lines of code

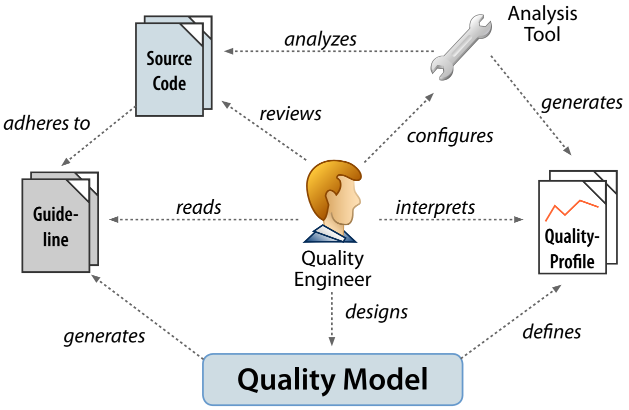


Figure 1. Model-Based Quality Controlling

3.1. Hierarchical Models

The idea of explicitly modeling maintenance activities was based on our experiences with building large hierarchical quality models. In turn of this process it became harder and harder to maintain a consistent model that adequately describes the interdependencies between the various quality criteria. A thorough analysis of this phenomenon revealed that our model and indeed most previous models mixed up nodes of two very different kinds: maintenance *activities* and *characteristics* of the system to maintain. An example for this problem is given in Fig. 2 which shows the *maintainability* branch of Boehm's *Software Quality Characteristics Tree* [2].

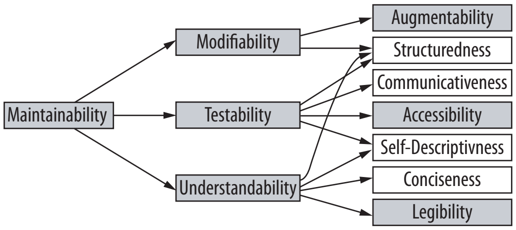


Figure 2. Software Quality Tree

Though (substantivated) adjectives are used as descriptions, the nodes in the gray boxes refer to activities whereas the uncolored nodes describe system characteristics (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 claim that this mixture of activities and characteristics is at the root of most problems encountered with previous models. The semantics of the edges of the tree is unclear or at least ambiguous because of this mixture. And since the edges do not have a clear meaning they neither indicate a sound explanation for the relation of two nodes nor can they be used to aggregate values!

As the actual maintenance efforts strongly depend on both, the type of system and the kind of maintenance activity, it should be obvious that the need to distinguish between activities and characteristics becomes not only clear but imperative. This can be illustrated by the example of two development organizations where 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 organizations will pay attention to some common attributes such as documentation, A and B would and should rate the maintainability of S in quite different ways because they are involved in fundamentally different *activities*.

Focusing on the individual factors that influence productivity within a certain context widens the scope of the relevant criteria. A and B's productivity is not only determined by the system itself but 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. To clarify that our observations are not limited to the software system itself, we speak about the *situation* instead of *software system* from now on. Similar to the software system that can be decomposed in *components*, the situation can be decomposed in, what we call *facts*.

3.2. An Activity-Based Model for Maintainability

The consequent separation of activities and facts leads to a new 2-dimensional quality model that regards *activities* and *facts* as first-class citizens for modeling maintainability.

The set of relevant activities depends on the particular development and maintenance process of the organization that uses the quality model. As an example, we use the IEEE 1219 standard maintenance process [10]. Its activity breakdown structure is depicted in Fig. 3. For the sake of brevity we only show a subset of the activities.

The 2nd dimension of the model, the facts of the situation, are modeled similar to an FCM model but without activity-based nodes like *augmentability*. It is important to understand, that we do not limit this dimension to properties of the software system, e. g. *structuredness*, but try to capture all factors that affect one or more activities. An excerpt of a facts tree is shown in Fig. 4.

Obviously, the granularity of the facts shown in the diagrams are too coarse to be actually evaluated. We follow the FCM approach in the situation tree by breaking down high level facts into detailed, tangible ones which we call *atomic* facts. An atomic fact is a fact that can or must be assessed

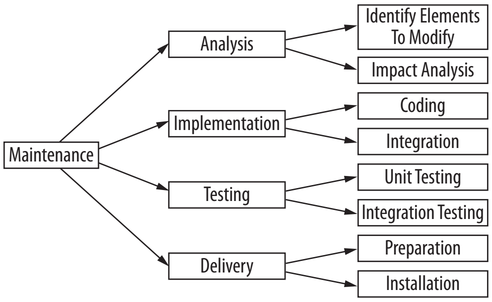


Figure 3. Example Activities

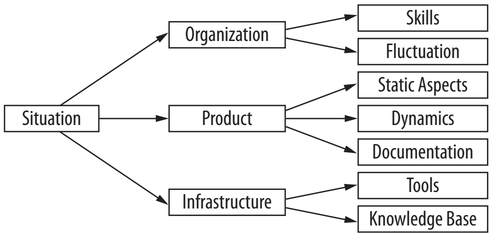
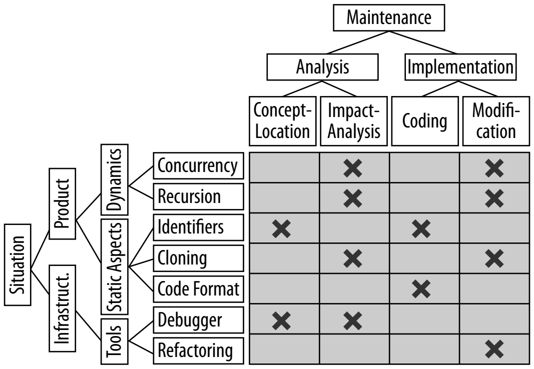


Figure 4. Example Facts

without further decomposition either because its assessment is obvious or there is no known decomposition.

To achieve or measure maintainability in a given project setting we now need to establish the interrelation between facts and activities. Because of the tree-like structures of activities and fact it is sufficient to link atomic facts with atomic activities. This relationship is best expressed by a matrix as depicted in the simplified Fig. 5.



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

Figure 5. Maintainability Matrix

The matrix points out what activities are affected by which facts and allows to aggregate results from the atomic level onto higher levels in both trees because of the unambiguous semantics of the edges. So, 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 facts and activities and therefore merely expresses the existence of a relation between a fact and an activity. To express different directions and strengths of the relations, more elaborate scales can be used here. Below, we show the application of a three-valued scale that proved to be sufficient for our current work.

The aggregation within the two trees provides a simple means to cross-check the integrity of the model. For example, the sample model in Fig. 5 states, that tools do not have an impact on coding, which is clearly nonsense. The problem lies in the incompleteness of the depicted model, that does not include tools like integrated development environments.

3.3. Attributes & Impacts

We found that a fine-granular decomposition of the situation (the facts tree) inevitably leads to a high number of repetitions as the same properties apply to different kind of artifacts. For example, *consistency* is obviously required for identifier names as well as for the layout of the documentation.

Therefore our model further decomposes facts into *entities* and *attributes* where entities "are the objects we observe in the real world" and attributes are "the properties that an entity possesses" [14]. Hence, entities describe a plain decomposition of the situation. Examples are *documentation*, *classes*, *variables* or the available *infrastructure*. Entities are associated with one or more attributes like *consistency*, *redundancy*, *completeness* or *superfluousness*.

So, the facts defined in the facts tree are actually tuples of entities and attributes: [Entity e | ATTRIBUTE A]. They describe properties of the situation that are desired or undesired in the context of maintainability. Examples are [Identifiers | CONSISTENCY], [Documentation | COMPLETENESS] or [Debugger | EXISTENCE] that simply describes the presence or absence of a debugging tool.

Note that the separation of entities and attributes does not only reduce redundancy but allows for a clean decomposition of the situation. This can be illustrated by an example of the quality taxonomy defined in [20]: *System Complexity*. As *System Complexity* appears too coarse-grained to be assessed directly, 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 subelement of *System Complexity* is. A separation of the entity and the attribute 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].

Impacts Using the notation introduced for facts we can elegantly express the impact a fact has on an activity with a three-valued scale where '+' expresses a positive and '−' a negative impact (the non-impact is usually not made explicit):

$$[\text{Entity } e \mid \text{ATTRIBUTE } A] \xrightarrow{+/-} [\text{Activity } a]$$

Examples are [Debugger | EXISTENCE] $\xrightarrow{+}$ [Fault Diagnostics], that describes that the existence of a debugger has a positive influence on the activity fault diagnostics. [Identifiers | CONSISTENCY] $\xrightarrow{+}$ [Concept Location] describes that consistently used identifier names have a positive impact on the concept location activity. [Variable | SUPERFLUOUSNESS] $\xrightarrow{-}$ [Code Reading] describes that unused variables hamper the reading of the code. To overcome the problem of unjustified quality guidelines each impact is additionally equipped with a detailed description.

Assessment Obviously, the facts are the elements of the model that need to be assessed in order to determine the maintainability (or maintenance effort) of a situation. Since many important facts are semantic in nature and inherently not assessable in an automatic manner, we carefully distinguish three fact categories:

1. Facts that can be assessed or measured with a tool. An example is an automated check for `switch`-statements without a `default`-case ([Switch Statement | COMPLETENESS]).

2. Facts that require manual activities; e. g. reviews. An example is a review activity that identifies the improper use of data structures ([Data Structures | APPROPRIATENESS]).

3. Facts that can be automatically assessed to a limited extent requiring additional manual inspection. An example is redundancy analysis where cloned source code can be found with a tool but other kinds of redundancy must be left to manual inspection ([Source Code | REDUNDANCY]).

3.4. The Quality Metamodel

Although most quality models conform to an implicitly defined metamodel they usually lack an explicitly specified metamodel that precisely defines the set of legal model instances. In contrast to this, our model is based on the explicit quality metamodel QMM. This metamodel consists of the elements discussed above: entities, attributes, facts, activities and impacts. For space reasons we do not explain this metamodel in detail but present a UML class diagram that illustrates the different model elements and their interplay (Fig. 6). Please note, that the figure shows only the core elements and omits details like the explanation texts that are associated with each element. Moreover, it does not show that the model features a generalization mechanism that allows attribute inheritance. It is, for example, possible to specify attribute SUPERFLUOUSNESS for entity Component and inherit it to entity Class.

The benefit of an explicit metamodel is twofold: First, it ensures a consistent structure of quality models. Second, it is a necessary basis for modeling tool support as described in the next section.

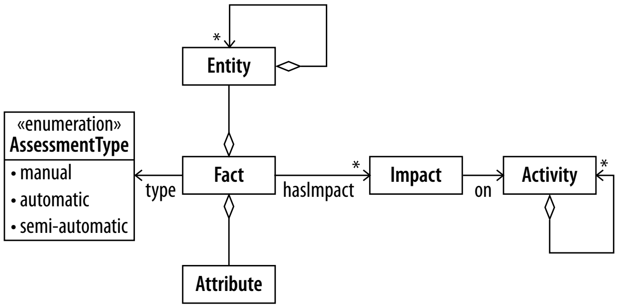


Figure 6. The Quality Metamodel QMM

3.5. Tool Support

Comprehensive maintainability models typically contain several hundred model elements. For example, the model that was developed for a commercial project in field of telecommunication [3] has a total of 413 model elements consisting of 160 facts (142 entities and 16 attributes), 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.

Due to the fact that our quality models are based on an explicit metamodel 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[3].

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. However, unlike classic, hand-written guidelines the automatically generated ones are guaranteed to be synchronized with the quality model that explicitly captures the understanding of quality within a project or a

---

[3] A beta version of the editor can be downloaded from `http://www4.cs.tum.edu/~ccsm/qmm/`

company. Guideline documents can be tailored to specific needs by defining selected views on the model. For example, a guideline document could be specifically generated to be used during documentation review sessions.

3.6. Summary

Our approach to modeling maintainability is based on the quality metamodel QMM. It advances on previous approaches to model maintainability and quality with respect to the following issues:

Focus on Activities Our model enforces a clear separation of system characteristics and maintenance activities. This separation makes activities, which constitute the main cost factor in software maintenance, first-class citizens in quality modeling and thereby contributes to a discussion of quality economics [24].

Unambiguous Decomposition Criteria Previous hierarchical models often exhibit a “somewhat arbitrary selection of characteristics and sub-characteristics” [12, 13]. We claim this due to the fact that previous models typically lack a clearly defined decomposition criterion. For example, it is not entirely clear how *self-descriptiveness* relates to *testability* in Boehm’s quality characteristics tree if one is not satisfied with a trivial “has something to do with”. Our approach overcomes this shortcoming by rigorously separating aspects that are typically intermingled: activities, entities and attributes. This separation creates separate hierarchies with clearly defined decomposition criteria.

Scope Our approach is not limited to modeling quality characteristics of a system itself. It includes external but equally important organizational issues like the existence of a configuration management process or the available tool infrastructure.

Explicit Metamodel Unlike other approaches known to us, our approach is based on an explicitly defined metamodel. This enables us to provide a rich set of tools for editing and maintaining quality models. Most importantly, the metamodel is a key required for the model-based quality controlling approach outlined before. Additionally, the metamodel fosters the conciseness, consistency and completeness of quality models as it forces the model designer to stick to an established framework and supports him in finding omissions. Examples are given in the next section.

4. Case Study

The applicability and usefulness of the approach described above was evaluated in a case study with a German truck and bus manufacturer. Specifically, we built a model for the maintainability of the Matlab Simulink and Stateflow models used for code generation. For this, the quality model we had developed in a project in the field of telecommunication was modified and extended with Simulink/Stateflow- specific elements. Guideline documents were generated from the model and automatic analyses were derived.

4.1. Environment

MAN Nutzfahrzeuge Group The MAN Nutzfahrzeuge Group is a German-based international supplier of commercial vehicles and transport systems, mainly trucks and busses. It has over 34,000 employees world-wide of which 150 work on electronics and software development. Hence, the focus is on embedded systems in the automotive domain.

The organization brought its development process to a high level of maturity by investing enough effort to redesign it according to best practices and safety critical system standards. The driving force behind this redesign was constantly focusing on how each activity contributes to global reliability and effectiveness. Most parts of the process are supported by an integrated data backbone developed on the eASEE framework from Vector Consulting GmbH. On top of this backbone, a complete model-based development approach has been established using the tool chain of Matlab/Simulink and Stateflow as modeling and simulation environment and TargetLink of dSpace as C-code generator. We describe the application and adoption of our model to this concrete situation and the generated benefits. The study lead to the adoption of the model into the MAN standard development process.

Embedded Systems and Matlab/Simulink Matlab/Simulink is a model-based development suite aiming at the embedded systems domain. It is commonly used in the automotive area. The original *Simulink* has its focus on continuous control engineering. Its counterpart *Stateflow* is a dialect of statecharts that is used to model the event-driven parts of a system. The Simulink environment already allows to simulate the model in order to validate it.

In conjunction with code generators such as Real-Time Workshop by MathWorks or TargetLink by dSpace it enables the complete and automatic transformation of models to executable code. This is a slightly different flavor of model-based development than the MDA approach proposed by the OMG[4]. There is no explicit need to have different types of models on different levels and the modeling language is not UML. Nevertheless, many characteristics

---

[4] `http://www.omg.org/mda/`

are similar and quality-related results could easily be transferred to an MDA setting.

4.2. The Maintainability Model

The initial maintainability model that was developed in the field of telecommunication (Sec.3) already covered various areas that we consider important for MAN, too. Examples for this are the parts of the model dedicated to architectural aspects or to the development process.

In the case study, we augmented the existing maintainability model with model elements that address Simulink/Stateflow-models, which are used as basis for code generation. Although such models are seemingly different from traditional source code, we found that a great number of source-code-related facts could be reused for them as they fundamentally serve the same aim: specify executable production-code.

Specifically, we extended the facts tree of the maintainability model with 87 facts (64 entities and 3 attributes) that describe properties of entities not found in classical code-based developed. Examples are states, signals, ports and entities that describe the graphical representation of models, e. g. colors. Furthermore, we modified the activities tree to match the MAN development process and added two activities (*Model Reading* and *Code Generation*) that are specific for the model-based development approach. 84 impacts describe the relation between facts and activities.

The newly developed parts of the maintainability model are based on three types of sources: (1) existing guidelines for Simulink/Stateflow, (2) scientific studies about model-based development and (3) expert know-how of MAN's engineers.

Specifically, our focus lies on the consolidation of four guidelines available for using Simulink and Stateflow in the development of embedded systems: the MathWorks documentation [16], the MAN-internal guideline, the guideline provided by dSpace [8], the developers of the TargetLink code-generator, and the guidelines published by the MathWorks Automotive Advisory Board (MAAB) [17].

Because of space and confidentiality reasons, we are not able to fully describe the MAN-specific model here. However, we present a number of convincing examples that demonstrate how our approach helps to overcome different kinds of shortcomings.

We start with a simple translation of the existing MAN guidelines for Stateflow models into the maintainability model. For example, the MAN guideline requires the current state of a Stateflow chart to be available as an measurable output. This simplifies testing of the model and improves the debugging process. In terms of the model this is expressed as [Stateflow Chart | ACCESSIBILITY] $\xrightarrow{+}$ [Debugging] and [Stateflow Chart | ACCESSIBILITY] $\xrightarrow{+}$ [Test].

We describe the ability to determine the current state with the attribute ACCESSIBILITY of the entity Stateflow Chart. The Stateflow chart contains all information about the actual statechart model. Note that we carefully distinguish between the *chart* and the *diagram* that describes the graphical representation. In the model the facts and impacts have additional fields that describe the relationship in more detail. This descriptions are included in generated guideline documents.

Consolidation of the Terminology In the case study we found that building a comprehensive quality model has the beneficial *side-effect* of creating a consistent terminology. By consolidating the various sources of guidelines, we discovered a very inconsistent terminology that hampers a quick understanding of the guidelines. Moreover, we found that even at MAN the terminology has not been completely fixed. Fortunately, building a quality model automatically forces the modeler to give all entities explicit and consistent names. The entities of the facts tree of our maintainability model automatically define a consistent terminology and thereby provide a glossary.

One of many examples is the term *subsystem* that is used in the Simulink documentation to describe Simulink's central means of decomposition. The dSpace guideline, however, uses the same term to refer to a *TargetLink subsystem* that is similar to a Simulink subsystem but has a number of additional constraints and properties defined by the C-code generator. MAN engineers on the other hand, usually refer to a *TargetLink subsystem* as *TargetLink function* or simply *function*. While building the maintainability, this discrepancy was made explicit and could be resolved.

Resolution of Inconsistencies Furthermore, we are not only able to identify inconsistencies in the terminology but also in contents. For the entity *Implicit Event* we found completely contradictory statements in the MathWorks documentation and the dSpace guidelines.

- *MathWorks [16]* "Implicit event broadcasts [...] and implicit conditions [...] make the diagram easy to read and the generated code more efficient."
- *dSpace [8]* "The usage of implicit events is therefore intransparent concerning potential side effects of variable assignments or the entering/exiting of states."

Hence, MathWorks sees implicit events as improving the readability while dSpace calls them intransparent. This is a clear inconsistency. After discussing with the MAN engineers, we adopted the dSpace view.

Revelation of Omissions An important feature of the quality metamodel is that it supports inheritance. This became obvious in the case study after modeling the MAN guidelines for Simulink variables and Stateflow variables. We model them with the common parent entity Variable that has the attribute LOCALITY that expresses that variables must have the smallest possible scope. As this attribute is inherited by both types of variables, we found that this important property is not expressed in the original guideline. Moreover, we see by modeling that there was an imbalance between the Simulink and Stateflow variables. Most of the guidelines related only to Simulink variables. Hence, we transferred them to Stateflow as well.

Integration of Recent Research Results Finally, we give an example of how a scientific result can be incorporated into the model to make use of new empirical research. The use of Simulink and Stateflow has not been intensively investigated in terms of maintainability. However, especially the close relationship between Stateflow and the UML statecharts allows to reuse results. A study on hierarchical states in UML statecharts [5] showed that the use of hierarchies improves the efficiency of understanding the model in case the reader has a certain amount of experience. This is expressed in the model as follows: [Stateflow Diagram | STRUCTUREDNESS] $\xrightarrow{+}$ [Model reading].

4.3. Usage of the Model

In the case study, we concentrated on checklist generation and some preliminary automatic analyses. Those were chosen because they promised the highest immediate payoff.

Checklist Generation We see quality models as central knowledge bases w.r.t. quality issues in a project, company, or domain. This knowledge can and must be used to guide development activities as well as reviews. However, the model in its totality is too complex to be comprehended entirely. Hence, it cannot be used as a quick reference. Therefore, we use the tool support for the quality model to select subsets of the model and generate concise guidelines and checklists for specific purposes.

Automatic generation of guideline documents was perceived to be highly valuable as the documents could be structured to be read conveniently by novices as well as experts. Therefore the documents feature a very compact checklist-style section with essential information only. This representation is favored by experts who want to ensure that they comply to the guideline but do not need any further explanation. For novices the remainder of the document contains a hyper linked section providing additional detail. Automatic generation enables us to conveniently change the structure of all generated documents. More importantly, it ensures consistency within the document which would be error-prone in hand-written documents.

Preliminary Automatic Analyses. As the model is aimed at breaking down facts to a level where they can be assessed and they are annotated with the degree of possible automation, it is straightforward to implement automatic analyses. So far, we have not fully exploited the possibilities but we are able to show that simple facts can be checked in Simulink and Stateflow models. For this, we wrote a parser for the proprietary text format used by Matlab to store the models. Using this parser we are able to determine basic size and complexity metrics of model elements like states, blocks, ... Moreover, we can use the parser to automatically identify model elements that are not satisfactorily supported by the C-code generator. By integrating these analyses in our quality controlling toolkit CONQAT [6] we are able to create aggregated quality profiles and powerful visualizations of quality data.

4.4. Discussion

The metamodel and the corresponding method for modeling maintainability proposed in Sec. 3 proved to be applicable to real development environments in the case study. After a short time, the 2-dimensional structure was accepted by the MAN engineers. Especially the model's explicit illustration of impacts on activities was seen as beneficial as it provides a sound justification for the quality rules expressed by the model. Moreover, the general method of modeling – that inherently includes structuring – improved the guidelines: Although the initial MAN guideline included many important aspects, we still were able to reveal several omissions and inconsistencies. Building the model, similar to other model building activities in software engineering [22], revealed these problems and allowed to solve them.

Another important result is that the maintainability model contains a consolidated terminology. By combining several available guidelines, we could incorporate the quality knowledge contained in them and form a single terminology. We found terms used consistently as well as inconsistent terminology. This terminology and combined knowledge base was conceived useful by the MAN engineers.

Although the theoretical idea of using an explicit quality metamodel for centrally defining quality requirements is interesting for MAN, the main interest is in the practical use of the model. For this, the generation of purpose-specific guidelines was convincing. We not only build a model to structure the quality knowledge but we are able to communicate that knowledge in a concise way to developers, reviewers and testers. Finally, the improved efficiency gained by automating specific assessments was seen as important. The basis and justification for these checks is given by the model.

5. Conclusion

Although maintainability is undisputedly considered one of the fundamental quality attributes of software systems, the research community has not yet produced a sound and accepted definition or even a common understanding what maintainability actually is. Substantiated by various examples we showed that this shortcoming is due to intrinsic flaws of current approaches to define, assess and improve maintainability. We showed that there is a need to make maintenance activities first class citizens in modeling maintainability due to their economical importance. This notion is captured by our two-dimensional quality metamodel which maps facts about a development situation to maintenance activities and thereby highlights their impact on the maintenance effort.

In a case study in the automotive domain we showed that our metamodel and the accompanying tools could be successfully used to build a comprehensive maintainability model for the development of embedded systems with Simulink/Stateflow. The construction of the model helped to define a consistent terminology and to reveal omissions as well as contradictions in existing quality guidelines. Long-term benefits are gained by the automatic generation of specifically-tailored guideline documents and the usage of automatic quality assessments. The study lead to the inclusion of the model into the MAN standard development process.

Our future work with MAN focuses on widening the scope of the automated quality assessments.

After first encouraging results with modeling *usability* [27], we currently use the quality metamodel to model other quality attributes like reliability and performance. Furthermore, we plan to use an integrated quality model for all relevant quality attributes. Our aim is to unify the currently used isolated approaches to quality to enable a holistic but systematic discussion of quality. We are convinced that this is an important step towards our final goal of a truly economically justified practice of quality engineering. A discussion of our goals can be found in [24] and [25].

References

[1] G. M. Berns. Assessing software maintainability. *ACM Communications*, 27(1), 1984.

[2] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod, and M. J. Merrit. *Characteristics of Software Quality*. North-Holland, 1978.

[3] M. Broy, F. Deissenboeck, and M. Pizka. Demystifying maintainability. In *Proc. of the 4th Workshop on Software Quality*. ACM Press, 2006.

[4] D. Coleman, D. Ash, B. Lowther, and P. W. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8), 1994.

[5] J. A. Cruz-Lemus, M. Genero, M. E. Manso, and M. Piattini. Evaluating the effect of composite states on the understandability of UML statechart diagrams. In *Proc. 8th Int. Conf. on Model Driven Engineering Languages and Systems*, 2005.

[6] F. Deissenboeck, M. Pizka, and T. Seifert. Tool support for continuous quality assessment. In *Proc. of the 13th IEEE Int. Workshop on Software Technology and Engineering Practice*, 2005.

[7] R. G. Dromey. A model for software product quality. *IEEE Trans. Softw. Eng.*, 21(2), 1995.

[8] dSpace. *Modeling Guidelines for MATLAB/ Simulink/ Stateflow and TargetLink*, 2006.

[9] M. Halstead. *Elements of Software Science*. Elsevier Science Inc., New York, NY, USA, 1977.

[10] IEEE. 1219 Software maintenance. Standard, 1998.

[11] ISO. 9126-1 Software engineering - Product quality - Part 1: Quality model. International standard, 2003.

[12] B. Kitchenham, S. Linkman, A. Pasquini, and V. Nanni. The squid approach to defining a quality model. *Software Quality Journal*, 6(3):211–233, Sept. 1997.

[13] B. Kitchenham and S. L. Pfleeger. Software quality: The elusive target. *IEEE Software*, 13(1), 1996.

[14] B. Kitchenham, S. L. Pfleeger, and N. Fenton. Towards a framework for software measurement validation. *IEEE Trans. Softw. Eng.*, 21(12):929–944, 1995.

[15] R. Marinescu and D. Ratiu. Quantifying the quality of object-oriented design: The factor-strategy model. In *WCRE 2004*. IEEE CS Press, 2004.

[16] The MathWorks. *Simulink Reference*, 2006.

[17] MathWorks Automotive Advisory Board. *Controller Style Guidelines For Production Intent Using Matlab, Simulink And Stateflow*, 2001.

[18] T. J. McCabe. A complexity measure. In *ICSE 1976*. IEEE CS Press, 1976.

[19] J. McCall and G. Walters. *Factors in Software Quality*. The National Technical Information Service, Springfield, VA, USA, 1977.

[20] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Proc. of the Int. Conf. on Software Maintenance*, 1992.

[21] M. Paulk, C. V. Weber, B. Curtis, and M. B. Chrissis. *The Capability Maturity Model: Guidelines for Improving the Software Process*. Addison-Wesley, 1995.

[22] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *Proc. 27th Int. Conf. on Software Engineering*, 2005.

[23] Sun Microsystems. *Code Conventions for the Java Programming Language*, 1999.

[24] S. Wagner. Using economics as basis for modelling and evaluating software quality. In *Proc. of the First Int. Workshop on the Economics of Software and Computation*. IEEE CS Press, 2007. To appear.

[25] S. Wagner and F. Deissenboeck. An integrated approach to quality modelling. In *Proc. of the 5th Workshop on Software Quality*. IEEE CS Press, 2007. To appear.

[26] J. Q. Wilson and G. L. Kelling. Broken windows. *The Atlantic Monthly*, 249(3), 1982.

[27] S. Winter, S. Wagner, and F. Deissenboeck. A comprehensive model of usability. In *Proc. of Engineering Interactive Systems*. Springer, 2007. To appear.