Demystifying Maintainability *

Manfred Broy
broy@in.tum.de

Florian Deissenboeck
deissenb@in.tum.de

Markus Pizka
pizka@in.tum.de

Institut für Informatik
Technische Universität München
Garching b. München, Germany

ABSTRACT

Due to its economic impact "maintainability" is broadly accepted as an important quality attribute of software systems. But in contrast to attributes such as performance and correctness, there is no common understanding of what maintainability actually is, how it can be achieved, measured, or assessed. In fact, every software organization of significant size seems to have its own definition of maintainability. We address this problem by defining an unique two-dimensional quality model that associates maintenance activities with system properties including the capabilities of the organization. The separation of activities and properties facilitates the identification of sound quality criteria and allows to reason about their interdependencies. The resulting quality controlling process enforces these criteria through tool-supported measurements as well as manual inspections. We report on our experiences with the incremental development of the quality model and its application to large scale commercial software projects. Among the positive effects are a slowdown of decay and a significantly increased awareness for long-term quality aspects.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.9 [**Software Engineering**]: Management—*Software quality assurance*

General Terms

Management, Measurement

Keywords

Maintainability, Quality Models, Quality Assessment

*Part of this work was sponsored by the German Federal Ministry for Education and Research (BMBF) as part of the project VSEK (www.software-kompetenz.de).

WOSQ'06, May 21, 2006, Shanghai, China.

1. ASSESSING MAINTAINABILITY

Virtually any software dependent organization has a vital interest in reducing its spending for software maintenance activities. This comes at no surprise as the bulk of the life cycle costs for software systems is not consumed by the development of new software but the continuous extension, adaption, and bug fixing of existing software [22]. In addition to financial savings, for many organizations, the time needed to complete a software maintenance task, such as an extension of an existing functionality, 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 economically manner becomes increasingly critical for numerous enterprises of diverse branches.

1.1 Myths

The term most frequently associated with more flexible software and significantly reduced long-term costs is *maintainability*. But what is maintainability?

Frequently found definitions of maintainability like "The effort needed to make specified modifications to a component implementation"[1] or "a system is maintainable if the correction of minor bugs only requires minor efforts" [25] seem overly simplified. In 2003 we conducted a study on software maintenance practices in German software organizations [16]. Interestingly, of the 47 respondents only 20% performed specific checking for maintainability during quality assurance. We further interviewed those 20% performing maintainability checking, about the criteria they used to check for maintainability. The individual responses differed significantly and ranged from object-orientation, cyclomatic complexity [19], limited numbers of lines per method, descriptive identifier naming, down to service oriented architectures or OMG's model-driven architecture.

From this we conclude, there is little common ground on what "maintainability" actually is, how it can be assessed, and how it could be achieved.

1.2 Consequences

One follow-up problem of this diffuse perception is closely associated with the term "legacy system". Many software systems that are deployed in large companies today are already 25 years or even older. Due to the absence of an

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

accepted set of criteria for the assessment of the maintainability of existing software systems and the low profile of structured assessment processes such as SRAH [23] they are often hastily coined as "legacy" because of rather inessential properties such as an unfashionable implementation language and style. The term "legacy" is often equated with being "unmaintainable" or the desire to replace it.

This in turn causes frequent reinventions of the wheel. We are indeed surprised how large organizations willingly spend enormous budgets just to replace existing "legacy" systems with new ones that frequently neither provide any increased "maintainability" nor added business value. In recent times, at least three large scale displacement projects have come to our attention, where the development of the new system was canceled even before their first release. The total loss of these three projects sums up to more than 50 million dollars.

To avoid such losses it is mandatory to substitute subjective judgement with solid reasoning by means of well-founded criteria that allow to assess the state of a system as well as the future impact of the defects it contains.

1.3 Well-Founded and Checkable

Certainly, programming and documentation guide lines as well as international standards [14] list various possible criteria for "maintainability". However, we argue that the missing impact and adoption of these criteria is due to one or both of the following two shortcomings of these criteria: first, being too general to be assessed (e. g. *modifiability*) or second, having no sound justification (e. g. methods may be no longer than 30 lines). Non-assessable criteria can inherently not have any impact, unjustified ones become ignored.

Therefore, effective criteria must be both well-founded and checkable. Otherwise, they will have no impact and/or will be ignored. Note that we stress "checkable" instead of "measurable with a tool" because we carefully distinguish between automatic, semi-automatic and manual checking (i. e. inspections) and exploit all three possibilities to effectively assess maintainability.

The approach presented in this paper uses a top-down method to identify criteria that fulfill these requirements. The stepwise top-down refinement of goals into subgoals and down to checkable criteria helps to achieve completeness and allows to reason about the criteria and their interplay. The starting point of this refinement is the breakdown of maintenance tasks into phases and activities according to [12]. Considering the diverse nature of activities, such as "problem understanding" and "testing" it becomes evident, that the criteria that actually influence the maintenance effort a numerous and diverse. Psychological effects, such as the *broken window* [24] deserve just as much attention as organizational issues (e. g. turnover) and properties of the code like naming of identifiers [5]. Any of these aspects may have a significant and vastly independent impact on the future maintenance effort.

2. RELATED WORK

Besides the rather vague definitions of maintainability cited above there are more elaborated definitions in the context of software metrics and quality modeling.

Metrics-based Approaches Several groups proposed metrics-based methods to measure attributes of software systems which are believed to affect maintenance [1, 4]. Typically, these methods use a set of well-known metrics like *lines of code*, Halstead volume [11], or McCabe's Cyclomatic Complexity [19] 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 economical facts of a software system, they still suffer from serious shortcomings. First, their intrinsic goal is to assess overall maintainability which is, as we claimed above, of questionable use as long as the organizational context of the observation and the future purpose of the system is ignored.

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.

Lastly, the indices and the underlying metrics are rarely validated and frequently violate the most basic requirements for measures; see measurement theory [10, 15]. 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 As maintainability is commonly perceived as a quality attribute similar to security or safety [14] it is only natural that research on software maintenance adopted many ideas from the broader field of software quality. A promising approach developed in this field are *quality models* which aim at describing complex quality criteria by breaking them down into more manageable sub-criteria. Such models are usually 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 some software metrics. The values determined by the metrics can then be aggregated towards the root of the tree to obtain values for higher level quality attributes. This method is frequently called the decompositional or *Factor-Criteria-Metric* (FCM) approach and was first used by McCall [20] and Boehm [3].

Although these and more recent approaches like [8, 7, 18] are clearly superior to the purely metrics based approaches described above, they have also failed to establish a broadly acceptable basis for quality assessments so far. We believe the reasons for this are the prevalent yet unrealistic desire to condense quality attributes as complex as maintainability into a single value and the fact that these models typically limit themselves to a fixed number of model levels. For example, FCM's 3 level structure is inadequate. High level goals like *usability* can not be broken down into measurable properties in only 2 steps. Further troublesome is their reluctance against properties that can not be measured automatically or aren't directly related to the product but the associated organization. E. g. it is incomprehensible why non of the models known to us highlights the influence of organizational issues like the existence of a configuration management processes on the overall maintenance effort.

Processes and Process Models Typically these organizational issues are 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 [17]. Of course, processes are of high importance and they determine the reliability and reproducibility of the development process. However, the quality of the outcome still strongly depends on the actual criteria, skills, and tools used during development.

There's 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. Nevertheless, there are significant gaps and severe misconceptions which demand to be filled respectively corrected.

3. MODELING MAINTAINABILITY

To provide a solid foundation for both, the assessment of existing systems and the development of new long-lived software systems we developed a two dimensional quality model that integrates and explains relevant criteria and describes their impact on actual maintenance activities. The following paragraphs describe the process that has led to this model and the final result.

3.1 Acts vs Facts

In an initial step we together with our industrial partners conducted several brainstorming sessions led by the question "What are the factors that influence maintenance productivity?". Our aim was to collect all relevant ideas and build a FCM like decompositional quality model from there. Unlike Dromey [8] suggested we did not build the model bottom-up starting from the measurable criteria. Instead, we tried to build the model top-down to ensure that all criteria considered relevant for maintenance productivity, independently from the question on how difficult the measurement or assessment could become, are collected.

In turn of this incremental refinement process it became harder and harder to maintain a consistent model that adequately described 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 figure 1 which shows the *maintainability* branch of Boehm's *Software Quality Characteristics Tree* [3].

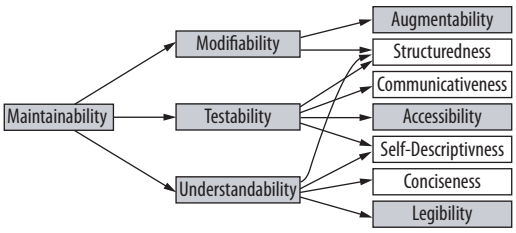

Figure 1: Software Quality Characteristics Tree

Though 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. On 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're 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 capture all these factors together with the system's characteristics we choose to speak about *facts about the situation* instead of properties of the software system from now on.

3.2 The 2-Dimensional Model

The consequent separation of activities and facts leads to a new 2-dimensional quality model that regards *activities* and *facts* as rows and columns of a matrix with explanations for their interrelation as its elements.

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 [13]. Its activity breakdown structure is depicted in figure 2. To the sake of brevity we only show a subset of the activities. Note, that the edges of the tree do have a clear meaning, that is the decomposition of activities into subtasks.

The 2nd dimension of the model, the facts about the situation, are modeled similar to an FCM model but without activity-based nodes like *augmentability*. It's 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 figure 3. Again, the semantics of the edges within this tree is free from ambiguity though different from the activity tree. The joint development of the facts tree with our industrial partners lead to more than 250 different facts.

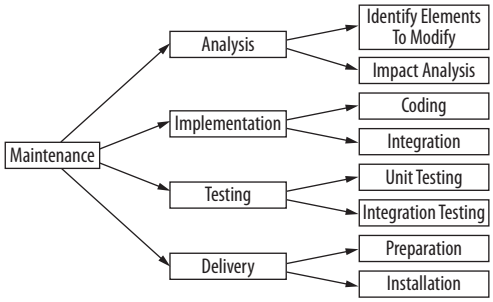

Figure 2: Example activities

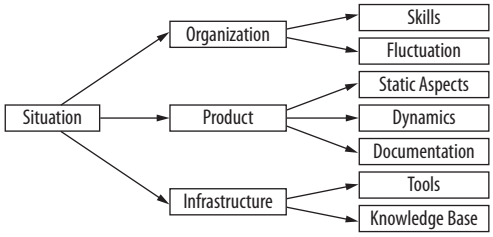

Figure 3: Example facts

Obviously the granularity of the facts shown in the diagrams are too coarse to be actually evaluated. We follow the FCM approach for the situation tree by stepwise refining high level facts into detailed, tangible ones which we call *atomic* facts. An atomic fact is a fact that can or must be assessed or checked without further decomposition either because its assessment is obvious or there is no known decomposition.

Since many important atomic facts are semantic in nature and inherently not computable, we carefully distinguish three fact categories for the implementation of the quality model.

1. Computable facts that can be extracted or measured with a tool. An example is an automated check for `switch`-statements without a `default`-case.

2. Facts that require manual activities; e. g. reviews. An example is a review activity that identifies the improper use of data structures.

3. Facts that can be computed 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.

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 facts it is sufficient to link atomic facts with atomic activities. This relationship is best expressed by a matrix as depicted in the simplified figure 4.

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 2 maintenance activities.

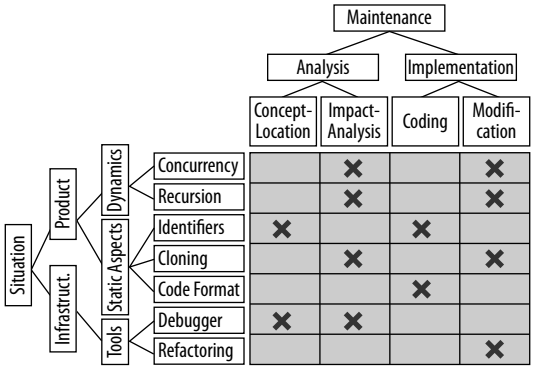

Figure 4: Maintainability Matrix

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

So far, we have only used a Boolean relation between facts and activities. This can be viewed as regarding every fact of equal importance for any activity which is of course untrue. To better reflect the different impact of the various facts we use relative weights as elements of the matrix. A relative weight is a value within the interval $[0, 1]$ and denotes the impact of the fact on the activity relative to the other facts. A value of 0 refers to no impact and 1 to a situation where the activity is affected by the corresponding fact, only. Consistency of weighting requires that the sums in each column (per atomic activity) is 1.

It should be evident, that the precise determination of the weights will require many years of research and empirical studies similar to the parameters in economic models like CoCoMo [2]. However, we made the experience that even rough estimates of experienced members of the team will already provide good starting points and provide guidance for development teams to focus on relevant quality facts (see section 6).

4. SAMPLE FACTS

As the instance of the model that we use in commercial projects is far too big to be presented in its entirety the following examples are meant to illustrate some details of a realistic instance of our two-dimensional model and emphasize the broad spectrum covered. Since the activity tree is individually determined by each organization according to its processes the following examples focus on the rather invariant facts tree that covers properties of the product as well as organizational issues.

4.1 Product Subtree

Clearly, most maintenance activities are somehow influenced by the software product that is to be maintained. Therefore, the bulk of the nodes of the facts tree is concerned with product characteristics. Important facts about the product are concerned either with the code itself or its documentation.

As shown in figure 3 the facts about the code are subdivided into facts about the static and dynamic structure of the code. This categorization was chosen because static as

well as dynamic aspects of a program are crucial for its comprehension [9] and comprehension is known to be of major importance during software maintenance [12, 1].

Another subtree captures clumsy or dangerous constructs and practices sometimes described as *bad smells* or *anomalies*. This includes amongst others code cloning, unused code, or unhandled exceptions. Again some of these facts can be checked automatically using tools and others need manual inspection.

Due to its importance, a considerable number of facts is devoted to documentation properties. Identified facts influencing the readability of the documentation include presentational issues as well as the content of the documentation. Facts of paramount importance are outdatedness, incoherence with the program, redundancy and inadequate breadth. Unfortunately, the documentation subtree offers very sparse opportunities for automated assessments and calls for manual reviews.

We experienced that the detailed explanation of important facts provided through the documentation subtree (currently about 40 nodes) greatly helped to structure the quality assurance process for documentation. Apart from that there's indeed a small but valuable number of facts which can be assessed in an automated way, e. g. for a Java-based system we use an automated check to make sure that all packages, classes, methods, and fields are commented (see also section 6).

4.2 Organization Subtree

As stated earlier, a model of maintainability may not restrict itself to product characteristics; it must take organizational issues into account as well. A drastic example that has come to our attention was a company that developed a product of fairly high quality but lost a significant part of its developers to a competitor. One can imagine that an incident like this dramatically increases the future maintenance effort though the system itself has not changed.

Obviously, maintenance productivity strongly depends on the people performing it. So one of the central organizational facts is concerned with *human resources* and contains atomic facts like turnover measured through the *annual turnover rate*. Another human resource fact is a *skill* node which may or may not be associated with productivity metrics.

Process-based research on software quality shows that well-defined processes do contribute to software quality. Our model instance does not go into details of software processes but checks for the existence of sub-processes like configuration management which undisputedly influences various maintenance activities.

A great deal of typical maintenance activities are nowadays efficiently supported by tools. Examples are debuggers, reengineering and visualization tools or configuration management tools. Due to the possible gains in productivity the use of such tools is crucial for efficient maintenance. So, our model instance features a *tool* subtree as part of the *organization* subtree. The *tool* subtree decomposes into respective facts about the different kinds of tools.

5. QUALITY CONTROLLING

This comprehensive and structured collection of criteria provides a precise specification of the required quality assurance activities and their frequency. The latter is due to fact that many defects, such as excessive redundancy because of copy&paste programming, can hardly be fixed later on but must be identified and eliminated as soon as possible. Criteria of this kind require daily checking while it is sufficient to check other criteria, such as the consistency of documentation and code, at certain points, only.

Manual quality controlling activities are inherently costly and must be substituted or supported with adequate tool support as far as possible. As a rule of thumb, the likelihood and frequency of a sophisticated quality assessments correlates with the availability of quality assurance tools.

Along with the model we developed the new quality assessment tool framework *ConQAT*. *ConQAT*'s flexible and reconfigurable pipes-and-filter-style architecture enables the composition of different assessment tools in a way that precisely matches our quality model. Details on *ConQAT* can be found in [6].

An example *ConQAT* output is shown in figure 5. Here, a simple traffic-light-scale is used to assess different criteria.

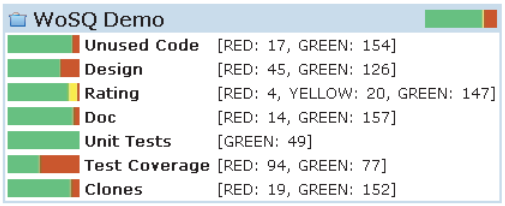

Figure 5: *ConQAT* Assessment Demo

6. EXPERIENCES

Our experiences with the two-dimensional model stem from a commercial project in the field of telecommunication. As the 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 very well suited for an application of our quality model.

We found that an efficient way of introducing the model was to present it as a single document in a guideline-like-fashion since this is what developers are familiar with. This linear representation of the model is generated from a relational database that stores all criteria, activities and their interrelationship. The guideline lists all atomic facts and their influence on the different maintenance activities, and developers can look up what facts influence which activities. In the appendix of the document developers find the relations between different facts. To keep things simple, the impact of atomic facts on the activities was modeled with a three-valued semantic: *negative influence*, *no influence*, *positive influence*.

At the beginning, we encountered a prevalent reluctance to maintainability assessments and were confronted with skepticism. However, developers and project managers alike soon started to develop an interest in maintainability issues after they realized the well-foundedness of the model. Since the matrix elements provide explanations for the influences of factors, their interdependencies, and their effects we could successfully foster a lasting discussion about quality and raise the awareness for the importance of quality issues in general and our quality model in particular.

[2]million lines of code

The model's other fundamental property, checkability, proved to be crucial for the acceptance of the model, too. Only after developers saw how they could actually assess a fact they were willing to accept its importance. Despite the fact that there is always a bias against facts demanding manual inspections, we found that the precise review guidelines provided by the model helped to motivate manual reviews.

Executives of the company regarded the application of different quality assessment tools and their integration within *ConQAT* for continuous quality controlling as the most substantial benefit of our endeavor. This view was shared by developers and project managers. Since the same organization failed to install a classic metric program due to lack of acceptance we claim this to be a noticeable success. We are convinced that this could only be achieved by presenting a guideline that not only lists a set of rules but provides a clear explanation of the relevant factors and their interdependencies.

Besides that the application of our model in an industrial project generated highly valuable insight into real-life aspects of software maintenance. On example are overly high *compilation times* that weren't included in our model in the beginning but seriously hamper productivity at the site of our industrial partner. A thorough analysis not only helped to understand the reasons for this problem but produced a solution that significantly reduced compile times.

7. 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 the intrinsic problem that there simply is no such thing as "the maintainability of a software system". We showed that the factors that influence maintenance productivity must be put into context with particular activities. This notion is captured by our novel two-dimensional quality model for software maintenance which maps facts about a development situation to maintenance activities and thereby highlights their relative influence.

Although the model is still incomplete and we don't claim completion is a task a single team of researchers can achieve, our experiences in a large commercial project not only support our work but generated measurable improvements alongside a plenitude of new insights.

Our current and future work focuses on the integration of these insights into the model and broadening the application of the model to other projects. The gathering of detailed data on the effects of the model will not only help to improve the model itself but gradually lead to a rich set of empirical data that will allow us to determine the relative weights of the various facts which is a prerequisite for developing accurate estimates for the benefit of quality improvements.

8. REFERENCES

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

[2] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

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

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

[5] F. Deissenboeck and M. Pizka. Concise and consistent naming. In *IWPC 2005*, pages 97–106, Washington, DC, USA, 2005. IEEE Computer Society.

[6] F. Deissenboeck, M. Pizka, and T. Seifert. Tool-supported realtime quality assessment. In *Pre-Proceedings of STEP 2005*, Budapest, Hungary, 2005.

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

[8] R. G. Dromey. Cornering the chimera. *IEEE Software*, 13(1), 1996.

[9] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *ICSM 2001*, 2001.

[10] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Trans. Softw. Eng.*, 20(3), 1994.

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

[12] C. S. Hartzman and C. F. Austin. Maintenance productivity. In *CASCON 1993*. IBM Press, 1993.

[13] *IEEE 1219 Software maintenance*. Standard, IEEE, 1998.

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

[15] C. Kaner and W. P. Bond. Software engineering metrics: What do they measure and how do we know? In *METRICS 2004*. IEEE CS Press, 2004.

[16] K. Katheder. Studie zur Software-Wartung. Bachelor thesis, Technische Universität München, Garching, Germany, Nov. 2003.

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

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

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

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

[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] T. M. Pigoski. *Practical Software Maintenance*. Wiley Computer Publishing, 1996.

[23] STSC. Software Reengineering Assessment Handbook v3.0. Technical report, STSC, U.S. Department of Defense, Mar. 1997.

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

[25] B. Wix and H. Balzert, editors. *Softwarewartung*. Angewandte Informatik. BI Wissenschaftsverlag, 1988.