How to effectively define and measure maintainability

Markus Pizka and Florian Deißenböck

Abstract

Maintainability and flexibility at the software level are of predominant importance to drive innovation at the business process level. However, existing definitions of maintainability, such as the Halstead Volume, McCabe's Cyclomatic Complexity or the SEI maintainability index provide a very poor understanding of what maintainability is how it can be assessed and ultimately controlled. This paper explains a new and more effective way to construct software product quality models. The key design principle is the strict separation of activities and properties of the system. This separation facilitates the identification of sound quality criteria and allows to reason about their interdependencies and their effects. The application of this quality modelling approach in large scale commercial software organizations helped to effectively reveal important quality shortcomings and raised the awareness for the importance of long-term quality aspects among developers as well as managers.

1. Introduction

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 are not consumed by the development of new software but the continuous extension, adaptation, and bug fixing of existing software [21]. 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. Maintainability – an ongoing confusion

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 such as "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" [24] are obviously overly simplified. The latter is particularly confusing as is in fact not a definition but a tautology. If one asked what a minor bug is, the answer would most certainly be "… its correction requires minor efforts". Besides these rather naive definitions, various metrics-based approaches try to define maintainability as compliance to a set of rules that correspond to measurable properties of the code, such as strong cohesion, limited coupling, etc. The general problem with this approach is the lack of a sound rationale for the selected criteria which in turn sometimes has a tendency of discussing some kind of technical beauty instead of effectively improving software maintenance.

In 2003 we conducted a study on software maintenance practices in German software organizations [15]. While 60% out of the 47 respondents said that they would consider

[1] SEI Open Systems Glossary

software maintenance as a “significant problem”, only 20% performed specific checking for maintainability during quality assurance. The criteria used by these 20% to check for maintainability differed significantly and ranged from object-orientation, cyclomatic complexity [18] limited numbers of lines per method, descriptive identifier naming, down to service oriented architectures or OMG's model-driven architecture.

Hence, there is little common ground on what “maintainability” actually is, how it can be assessed, and how it could be achieved.

1.2. “Maintainability” is misleading

This confusion can easily be explained and resolved by considering “maintainability” as a term. The “ility” ending is used to transform the adjective “maintainable” into a noun and thereby denote it as a property of a system. The adjective “maintainable” in turn denotes the assumption that the activity “to maintain” (verb) is a property of the object that we regard, i.e. a software system. However, the perception that the ability to maintain a system was a property of the system is very limited and neglects various other factors that have a strong influence on software maintenance activities, such as the qualification of maintainers, organizational knowledge management and adequate tools. This shortcoming is most obvious when it comes to “readability” since the ability to read is primarily not a property of the document or program to read but a question of the skills of the reader.

Therefore, we strongly argue that maintainability is not solely a property of a system but touches three different dimensions:

1. The skills of the organization performing software maintenance
2. Technical properties of the system under consideration
3. Requirements engineering

In addition to skills and technical properties as discussed above, dimension 3 – requirements engineering – plays an important role in defining “maintainability” because the question how much flexibility one wants to have at what points in a software systems for which purpose sets the goals for constructing and later on assessing the required flexibility of the software system. For example, the strategy to construct the shortest possible implementation without any flexibility and restructuring or even rewriting the system in the event of significant changes might yield lower maintenance effort than designing for flexibility in-advance with numerous levels of indirection that might never be needed [25].

1.3. Setting the goal – cost-effectiveness

From a practical point of view, the ultimate goal of any effort to improve software maintenance practices has to be increasing cost-effectiveness. While the questions whether certain architectural styles or coding and documentation techniques comply with a certain standard, or are up-to-date should be rather irrelevant, the question how to minimize the time and budget needed per change request should be of paramount importance. Throughout this paper, we therefore assume that “maintainability” requirement is to be cost-effective.

The quality model that we will introduce below respects the three different dimensions of maintainability as described above and is aligned to achieving cost-effectiveness by explicitly modelling maintenance activities which are the main drivers of software maintenance costs and putting them into relation with technical and organizational properties.

For the reminder of this paper, we will keep cost-effectiveness in mind as the requirements dimension of maintainability and focus on finding the important technical and organizational properties (i.e. dimensions 2 and 1) that influence software maintenance productivity.

1.4. Effective technical and organizational criteria

Certainly, programming and documentation guide lines as well as international standards [13] list various possible criteria for the technical dimension of ”maintainability”. However, the missing adoption of these criteria is due to one or both of the following two shortcomings of these criteria:

1. too general to be assessed (e.g. modifiability) or
2. no sound justification (e.g. methods may not be 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. Note that we stress “checkable” instead of “measurable with a tool” since we carefully distinguish between automatic, semi-automatic and manual checking (i.e. inspections) and exploit all three possibilities.

The approach presented in this paper uses a top-down method to identify criteria that fulfil 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 [11]. Considering the diverse nature of activities, such as “problem understanding” and “testing” it becomes evident, that the technical and organizational criteria that actually influence maintenance effort a numerous and diverse. Psychological effects, such as the *broken window* [23] deserve just as much attention as organizational issues (e.g. turnover) and properties of the code like naming of identifiers [5]. Each of these aspects has a significant impact on maintenance activities and therewith future maintenance costs.

2. Related work

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* [10], or McCabe's Cyclomatic Complexity [18] and combine them into a single value, called *maintainability index*.

Although such indices may expose a correlation with economical experiences, they suffer from serious shortcomings. First, their intrinsic goal is to assess overall maintainability which is, as we claimed above, of questionable use. Second, they limit themselves to properties that can be measured automatically by analyzing source code. 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. Others important technical properties such as useful documentation and normalized data modes are outside the scope of code analysis. Lastly, the indices and the underlying metrics frequently violate the most basic requirements of measurement theory [9] [14]. Because of this, well-known metrics, such as the Cyclomatic Complexity, are neither sufficient nor necessary to indicate a quality defect.

As the technical dimension of maintainability is a quality attribute of a system similar to security or safety [13] research on software maintenance adopted many ideas from the broader field of software quality. **Quality models** aim at describing complex quality criteria by breaking them down into more manageable sub-criteria. Such models usually organize quality attributes in a tree with an abstract quality attributes like “maintainability” at the top and more concrete ones like “analyzability” or “changeability” on lower levels. The leaf factors are ideally detailed enough to be objectively assessed. The values determined by the metrics are then aggregated towards the root of the tree to obtain values for higher level

quality attributes. This method is often called the decompositional or *Factor-Criteria-Metrics* (FCM) approach and was first used by McCall [19] and Boehm [3]. Although these and more recent approaches [7][8][17] are superior purely metrics based approaches, they also fail to establish a broadly acceptable basis for quality assessments. The reasons for this are the prevalent yet unrealistic desire to condense complex quality attributes 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 two 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 none 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.

Organizational issues are typically covered by **process-based approaches** to software quality like ISO 9000 or CMMi [20]. Albeit, the underlying assumption that good processes entail high quality products is a widely disputed misconception [16]. Well-defined processes are help to achieve reliability and reproducibility in software projects. However, the quality of the outcome still strongly depends on the actual skills, tools and criteria used during development.

3. Modelling criteria and their effects

To provide a solid foundation for assessing "maintainability" we developed a two dimensional quality model that integrates and explains relevant technical and organizational criteria and describes their impact on actual maintenance activities. The following paragraphs describe the rationale and the structure of the model. Further details are found in [28].

3.1. Acts and facts need to be separated

In an initial step we tried to answer the question "What are the factors that influence maintenance productivity?" by collecting relevant ideas from related work and building a FCM like decompositional quality model from there. Unlike Dromey [7] we did not build the model bottom-up starting from the measurable criteria but 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, were collected.

The incremental refinement of these factors showed that preserving a consistent model that adequately described the interdependencies between the various quality criteria became soon very hard. The reason for this was that our model just like other well-known other models mixed up nodes of two very different kinds: *activities* and *characteristics of the system*. An example for this problem is shown in figure 1 which shows the *maintainability* branch of Boehm's *Software Quality Characteristics Tree* [3].

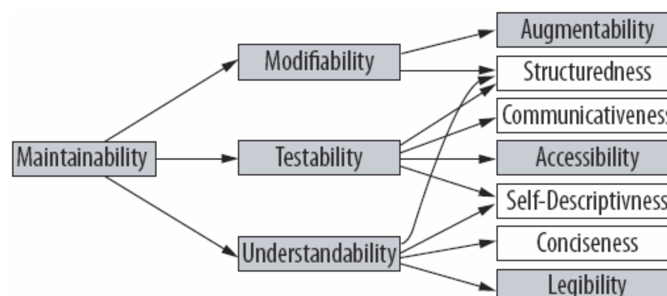

Figure 1 - *Software Quality Characteristics*

Though adjectives are used as descriptions the nodes in the gray boxes refer to activities whereas the uncolored nodes describe system characteristics. So the model should rather read as: When we *maintain* a system we need to *modify* it and this activity of *modification* is somehow influenced by the *structuredness* of the system. While this 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 known quality 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 is imperative.

3.2. Acts facts matrix

The separation of activities from facts leads to a two-dimensional model that regards activities and facts as rows and columns of a matrix with explanations for their interrelation as its elements.

The selection of activities depends on the particular development and maintenance process of the organization that uses the quality model. Here, we use the IEEE 1219 standard maintenance [12] as an example. An excerpt from its activity breakdown structure is shown in figure 2a. Now, the edges of the activity tree have the clear meaning of task composition. The 2nd dimension of the model, the facts about the situation, are modelled similar to an FCM model but without activity-based nodes like *readability* (2b). Again, the semantics of the edges within this tree is unambiguous though different from the activity tree.

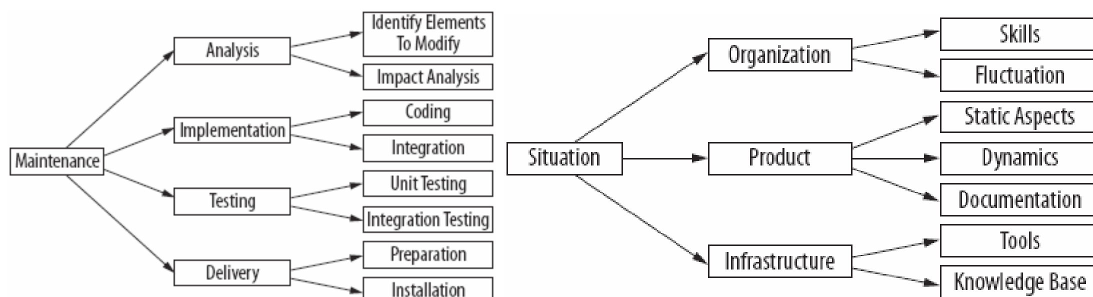

Figure 2 - *Example activity (a) and fact (b) trees*

Obviously, the granularity of the facts shown in the example is too coarse for proper evaluation of the facts. In practice, we refine the situation tree stepwise down to detailed, tangible facts that we call *atomic*. Since many important atomic facts are semantic in nature and inherently not computable, we carefully distinguish three fact categories.

1. Computable facts that can be extracted or measured with a tool.
2. Facts that require manual inspection.
3. Facts that can be computed to a limited extent requiring additional manual inspection. One example for this is dead code analysis.

The interrelation between atomic facts and activities can be best expressed by a matrix as depicted in the simplified figure 3. The matrix points out how facts affect activities (here simplified as true/false), allows to aggregate results from the atomic level onto higher levels in both trees because of the unambiguous semantics of the edges, and also allows to cross-check the integrity of the model. For example, the sample matrix states that tools don't affect

coding, which is untrue and due to the incompleteness of the example, that doesn't regard tools like modern integrated development environments.

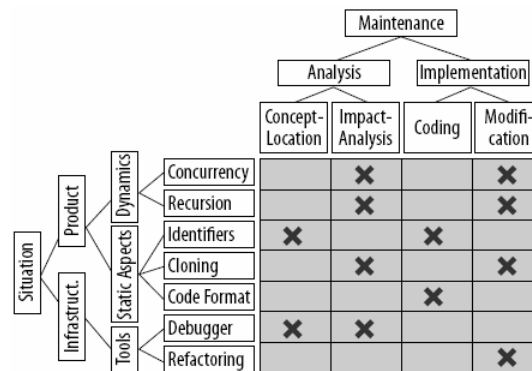

Figure 3 - *Matrix model explaining mainenance effort*

4. Experiences

The above described meta model was first designed and used to develop a quality model within a large scale commercial project in the field of telecommunication. The system regarded and assessed in this context consisted of 3.5 MLOC[2] in C++, COBOL and Java, was 15 years old and under active development by 50-100 developers which completed 150 change requests per year. The quality model was later-on refined, extended and used by three further large scale and well-known industrial organization. One of these organizations now uses this model as the foundation to design its company-wide coding and documentation rules [26]. The other organization used it to assess the state of a significant part of its core software landscape consisting of more than 10 MLOC of legacy software written in COBOL and PL/I.

Due to confidentiality requirements detailed results of these practical experiences can not be disclosed. However, we are able to summarize important results and describe selected aspects of the concrete model instances used in these practical settings to provide further information on the practical applicability of the model.

4.1. Relevant activities

As can be expected, the task to derive the activity tree is usually merely simple and is best accomplished by transforming the software development and maintenance process of the organization with its task, subtask and activity structure into an integrated tree representation. The top most node is usually either “software development” or “software maintenance”. From there it splits into the typical tasks: analyze, design, implement, test (subtasks “prepare testenvironment”, “perform test case”, “interpret test results”, deploy (“build”, “integrate”, “install”, “run”), document, and operate.

While all of these activities can easily be further refined to more detailed activities from a technical point of view, the reasonable refinement is in practice limited by the possibility to monitor the actual effort. The maximum useful refinement is predetermined by the possible entries of the time recording system. The impact of changing a fact in the situation on a certain activity can not be tracked if the same activity is not recorded individually in the time recording system. Activities of this kind can only be used as a rationale for selecting a situation fact. In turn, the matrix model has helped to identify weaknesses in time recording systems and has already fostered more detailed time recording systems some organizations paving the ground for a goal-oriented and truly economics driven optimization.

[2] Million Lines of Code

4.2. Facts about the situation

In all scenarios where the quality model was used, so far, the situation tree grew to at least 250 different facts about the situation which sounds large and complicated but is actually not surprising because the productivity of software maintenance activities is indeed severely influenced by various different aspects. Due to space constraints, we can only mention few of them to give some examples.

First of all the quality models used in practice regard not only the source code but also organizational aspects (skills, distribution of knowledge within the team, turnover, tools, structured processes, roles, test-suite, coding and documentation rules, etc.), the documentation of the system, the data model and even the actual data that is stored in the information system. The latter is usually neglected in other quality models but of high relevance. When performing changes on a large scale system, one has to ensure that the change is not only compatible with the existing code but also with the existing (legacy) data. For example, even if certain data violates new assumptions, such as a minimum or a maximum allowed time-stamp, the existing data either needs to be transformed or the new functionality must be prepared for possible exceptions to its rules. In both cases, the legacy data significantly increases the time need for the activities "impact analysis", "design", "implementation", and eventually "deploy".

Similar to that, we and our industrial partners found various aspects of data models of particular importance. We therefore introduced metrics about the data model such as max. number of primary key attributes per table, max. number of attributes per table and number of entries per table. In practical settings these facts proved to be highly useful to detect duplicated attributes (e.g. in a table with more than 150 attributes), lack of normalization and further shortcomings that negatively influence various activities.

Among the many important facts of the source code sub tree of the situation tree, we found two facts as extremely helpful. First is the code redundancy, second is IF-ratio – i.e. the number of simple conditionals ("IF") per 1000 lines of code (kLOC). We argue that redundancy duplicates the effort needed for analysis, implementation and testing. As described in [27], we consider redundancy as a main cost driver in software maintenance. In our practical work, we frequently found systems with 40% redundancy but also encountered systems with 90% redundancy! The IF-ratio proved to deliver a useful indicator for the algorithmic model behind a software system respectively its design. While well-structured systems typically expose a rate of 20-30 IFs per kLOC, we detected in legacy systems values of 60 and up to 100 IFs per kLOC. The main reason detected for these striking values were consistently change requests that were performed under strong pressure of time by simply adding new cases and exceptions to existing algorithms. The consequence of this is that analyzing and testing such systems becomes extremely difficult and time-consuming.

4.3. Using the quality model for assessments

Using the proposed quality model for assessing the state of large-scale software systems has proved to yield valuable information for both technical staff as well project and product managers. In all cases, the unusual criteria used in the model revealed severe weaknesses that were not regarded before, e.g. 90% redundancy or an average 20 times repetition of constant values. In addition to this, the integration of the two dimensions activity and situation in this model provided a common ground for members of the technical staff and manager to identify possible optimization. The model allows developers to explain the commercial impact of a property of the organization or the system to managers. In turn, manager use the model to identify the facts about the situation the drive unwanted costs of certain activities.

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 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. This model has been successfully applied in industrial settings to reveal significant economic optimization potential.

6. 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 et al. 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. Deißenböck and M. Pizka. Concise and consistent naming. In IWPC 2005, pages 97–106, Washington, DC, USA, 2005. IEEE Computer Society.
[6] F. Deißenböck, 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] N. Fenton. Software measurement: A necessary scientific basis. IEEE Tran. Soft. Eng., 1994.
[10] M. Halstead. Elements of Software Science. Elsevier Science Inc., New York, NY, USA, 1977.
[11] C. S. Hartzman, C. F. Austin. Maintenance productivity. In CASCON 1993. IBM Press, 1993.
[12] IEEE 1219 Software maintenance. Standard, IEEE, 1998.
[13] ISO 9126-1 Software engineering - Product quality - Part 1: Quality model. ISO, 2003.
[14] 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.
[15] K. Katheder. Studie zur Software-Wartung. Bachelor thesis, TU München, Germany, 2003.
[16] B. Kitchenham and S. L. Pfleeger. Software quality: The elusive target. IEEE Software, 1996.
[17] R. Marinescu and D. Ratiu. Quantifying the quality of object-oriented design: The factor-strategy model. In WCRE 2004. IEEE CS Press, 2004.
[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 (NTIS), Springfield, VA, USA, 1977.
[20] M. Paulk, C. V. Weber, B. Curtis, and M. B. Chrissis. The Capability Maturity Model: Guidelines for Improving the Software Process. Addison-Wesley, 1995.
[21] T. M. Pigoski. Practical Software Maintenance. Wiley Computer Publishing, 1996.
[22] STSC. Software Reengineering Assessment Handbook v3.0. Technical report, STSC, U.S. Department of Defense, Mar. 1997.
[23] J. Q. Wilson and G. L. Kelling. Broken windows. The Atlantic Monthly, 249(3), 1982.
[24] B. Wix and H. Balzert, editors. Softwarewartung. Angewandte Informatik. BI Wissenschaftsverlag, 1988.
[25] M. Fowler. Who needs an Architect? IEEE Software, pages 11-13, 20(5), September 2003.
[26] F. Deißenböck, S. Wagner, M. Pizka, S. Teuchert, J.-F. Girard. An Activity-Based Quality Model for Maintainability. Submitted to Int. Conf. on Software Maintenance, Paris, 2007.
[27] M. Pizka. Code Normal Forms. NASA SEW-29. Greenbelt, MD. April, 2005.
[28] M. Broy, F. Deißenböck, M. Pizka. Demystifying Maitainability. Proceedings of the 4th Workshop on Software Quality. ACM Press. Shanghai, China. 2006