Towards an Architecture Maintainability Maturity Model (AM^3)

Christoph Rathfelder, Henning Groenda

FZI Forschungszentrum Informatik,
Software Engineering,
Haid-und-Neu-Straße 10-14, 76131 Karlsruhe
{rathfelder, groenda}@fzi.de

Abstract

The maintainability of software systems is a crucial point in the software lifecycle. However, assessing the quality of the software's architecture with respect to evolution is a challenging task. The evaluation of the maintainability of a system's architecture is often made using scenario-based techniques. These techniques require a comprehensive anticipation of future adaptations of the systems. To circumvent this problem, a scenario-independent method is desirable to assess maintainability. Additionally, the comprehensibility of the architecture for third persons which were not involved in the initial design is an important aspect in the long-term. We therefore developed the Architecture Documentation Maturity Model (AM^3) to assess the quality of the architecture's documentation as this first of all influences comprehensibility. This model is a first step towards a more general approach to assess the maintainability of architectures, called Architecture Maintainability Maturity Model.

1 Introduction

Maintainability is one of the most important quality attributes of a software system in the long term. Evolution becomes necessary, for example if enterprises want to adapt their software systems to changed business needs or integrate systems after a merger. Although evolution of software systems is quite common, it is hardly possible to make general quantifiable statements about the maintainability of a system. The effort to maintain a system on a large scale is affected by its architecture, especially if a restructuring of the system is required (e.g, additional functionality is required). On a small scale, the implementation itself influences the maintainability of the system, for example if bugs have to be detected and fixed or the performance of critical parts is optimized.

Nowadays, scenario-based approaches like the Architecture-Level Modifiability Analysis (ALMA) [4] or the Architecture Trade-Off Analysis Method (ATAM) [13] are the most widely used techniques to evaluate the maintainability of a system. These approaches require the definition of scenarios that represent the envisioned evolution of the system. Especially if the system has a planned lifetime of several years, the uncertainty is quite high and it is hardly possible to anticipate all needed adaptations of the system. A scenario independent technique is therefore desirable.

Another common method to assess the quality of software is the application of maturity models. Maturity models consist of different levels expressing the rating of the system under scrutiny regarding a certain evaluation aspect. The classification according to the different levels of a model is based on the assessment of indicators, which are defined for each level. The most popular maturity model for software is the Capability Maturity Model Integration (CMMI) [14]. It focuses on the quality of software development processes in whole enterprises.

A large part of the maintenance effort is induced for understanding the basic structure of a system and comprehend the underlying design decisions. The effort to comprehend a system's architecture thereby heavily depends on the knowledge of employees about the architectural decisions. Employees who should change a system but were not involved in its design and implementation need to familiarize themselves with the system and thereby have to rely on the quality of the documentation of design decision to make the right decision for themselves [17]. Hence, the documentation of the architecture has a large impact on its maintainability.

This paper presents the first step towards an Architecture Maintainability Maturity Model (AM^3). The aim of the AM^3 is the definition of indicators and maturity levels to allow reasoning about the maintainability of a system's architecture. Its maturity levels should be shaped accord-

ing to an ascending maintainability, meaning that a higher maturity level correlates with a better maintainability of the system's architecture.

The contribution of this paper is the definition of the Architecture Documentation Maturity Model (ADMM), as first part of the planned AM^3. The ADMM defines five maturity levels. Each maturity level represents an extension of the previous level in terms of the documentation quality. The accessory requirements on the architecture's documentation induce higher comprehensibility of the architecture which should lead to lower maintenance effort in the long-term.

This paper is structured as follows: Section 2 gives an overview of related work. Section 3 presents a definition of architecture maintainability. Section 4 describes the ADMM and its five maturity levels. Section 5 concludes the paper and provides an outlook to future work.

2 Related Work

Regarding the maintainability of software there are several definitions available. The IEEE standard glossary of software engineering terminology [10] defines maintainability as:

> The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.

The ISO/IEC 9126 standard [11] provides a complete quality model that breaks the quality of software systems down into the sub-characteristics functionality, reliability, usability, efficiency, maintainability, and portability. The ISO/IEC 9126 quality model defines maintainability as:

> The capability of the software to be modified.

The quality model emphasizes the following attributes that make up the maintainability of a system:

- **Stability:**
  The capability to avoid that modifications cause unexpected effects on other parts of the software system.
- **Analyzability:**
  The capability needed to search out deficiencies and causes of failures within the system.
- **Changeability:**
  The capability to extend, enhance, and customize a software system.
- **Testability:**
  The property of the software system to be tested effectively in order to observe and check the behavior of the system.

Both definitions of maintainability presented above are applicable for software systems in general and not specialized to a certain type of software system (e.g. component-based software systems (CBSS)). However, this generality also has its drawbacks as properties of certain system types are not exploited. For example, Grover et al. [7] emphasize that CBSS needs a refined definition of maintainability to reflect that maintenance of a CBSS more often requires a reconfiguration than a recoding of the system. This means that components are rather rearranged or substituted by other components than their implementation changed.

The Software Maintenance Maturity Model (SM^{MM}) [1] developed by April et al allows the evaluation of the maintenance activities and the determination of their maturity. Comparable to the CMMI, they assess the process with the aim to draw conclusions about attributes of the product from the process maturity. For this reason, the SM^{MM} does not consider the product itself and its documentation. It thereby gives no assistance in increasing the maintainability of a system or even its architecture. An additional maturity model with relation to maintainability is the Corrective Maintenance Maturity Model (CM^3) [12]. In contrast to the SM^{MM}, it evaluates the capability of an enterprise to perform maintenance activities and not the executed activities itself. It is focused on the knowledge and training of maintenance engineers and hence it does not provide indications to evaluate the maintainability of a software system.

Huang and Tilley [9] have described their idea of an Documentation Maturity Model (DMM). The DMM bases on the five maturity levels introduced by the CMMI. It does not distinguish between documentation of code and the description of the architecture. In contrast to ADMM, the DMM does not enjoin the information that has to be included within the documentation. It rather describes the techniques that have to be used within the documentation. Level 3 for example requires animated graphics and hyperlinks. Furthermore, a documentation of the system's architecture is not required until level 3.

3 Architecture Maintainability

Regarding CBSS, it has to be differentiated between maintainability on the *architecture level* and the *component level* [15]. The general definitions of a software system's maintainability do not consider this differentiation. For this reason, this section presents our definition and refinement of architectural maintainability. We use the definition of software architecture provided by Bass et al:

> "The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." [2]

Figure 1 provides an overview about the attributes of an architecture which affect its maintainability. This refinement of architecture maintainability is described in the following.

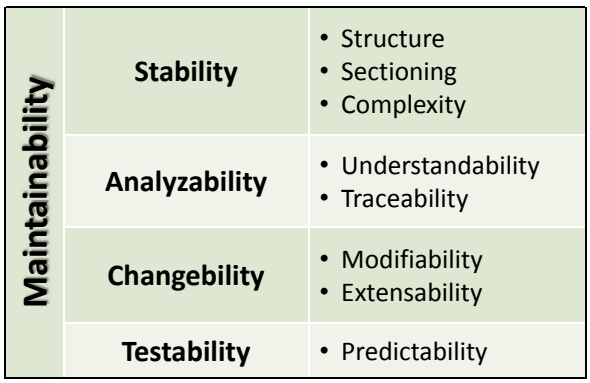

Figure 1. Architecture Maintainability

Following Szyperski's definition of a component [19], components are independently deployable and should not have any unexpected effects on other components. Hence, modifications of the architecture, in particularly the substitution of components, should not affect the functionality of other components. Nevertheless, extra-functional quality attributes (e.g., performance or availability) of the system and other components can be affected by such changes. The *stability* of an architecture is mainly formed by the ease to detect or even prevent such unwanted side-effects. This detection can be simplified by the structuring and sectioning of the architecture. Furthermore, the complexity of an architecture, which is also affected by the structuring and sectioning, impacts the ease to identify side-effects without testing the system.

Detecting and analyzing problems caused on the architectural level is a quite complex activity. Therefore, the *analyzability* of an architecture stands for the capability that design decisions made and the resulting architecture itself can be easily understood by experts which were not necessarily involved in the design process. The understandability of the architecture can for example be increased by a reasonable documentation or the usage of well-known design patterns. The second important aspect of analyzability is traceability. This means that architectural design decisions made with respect to stipulated requirements should be associated with these requirements in order to make decisions comprehensible for third persons.

Changeability is the most important attribute of maintainability on the architectural level. It has to be differentiated between modifiability, extensibility, and portability [15]. Modifiability stands for the capability of the architecture to be restructured in order to meet new or changed requirements (e.g. a shorter response time) of the system. Thereby, a modification of the architecture does not mean that new functionality is added to the system. It is rather a restructuring of the architecture. The main difference between modifiability and extensibility is that the later one targets the capability of the architecture to be extended with new components to realize additional functionality. In contrast to the aforementioned attributes modifiability and extensibility, portability means the capability of the later system to be adapted to other environment (e.g. another operating system or framework).

Testability of an architecture refers to the possibility to examine the behavior of the system. For this reason, testability of the architecture is not restricted to integration and system-level testing but also includes extra-functional reasoning techniques. Existing approaches that support the second kind of testability are for example SOFA [8] and Palladio [3].

4 Architecture Documentation Maturity Model

This section describes the developed ADMM and its maturity levels. The ADMM can be used to assess the quality of the description and documentation of a software system's architecture. There are 5 different maturity levels which are associated in an ascending order with an increasing number of requirements on the architecture's description and documentation. Growing maturity of the documentation thereby comes along with an increasing maintainability. A higher maturity of the documentation is in general accompanied by the use of more formal models as these provide a fixed semantic meaning of the modeled architectural artifacts. Furthermore, the use of formal models eases (semi-)automated analyses which can for example be used to estimate the impacts caused by modifications within the architecture. There is also the possibility to use model-driven techniques to provide a consistent documentation by automatically propagating changes in the architecture to the implementation and vice versa.

A precondition of the ADMM is that some kind of architectural description is already available. Systems without any architectural documentation therefore cannot be ranked in the ADMM. However, they can be subsumed as a virtual level 0 to point out their immaturity. The five maturity levels of ADMM are sketched in figure 2 and described in the following in more detail.

4.1 Level 1: Drawn

On this maturity level, simple graphics are used to describe the architecture. For example tools like Microsoft PowerPoint provide these drawing capabilities. The use

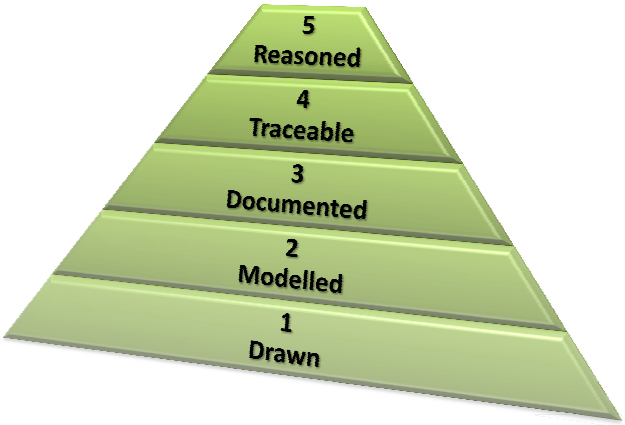

Figure 2. Maintainability Maturity Levels

of graphics for architectural descriptions eases the understanding of the architecture and can be used to provide an abstract overview of the architecture. The main disadvantage of these graphics is the fact that the used symbols have no specified semantic. Due to this lack of semantics, the graphics are often misinterpreted, which may in the worst case lead to inconsistent or wrong modifications of the architecture. Another common problem is to keep the documentation and implementation consistent. As there are no constraint checks on the graphics, the probability to produce inconsistent views within the documentation or between the documentation and implementation is quite high.

Nevertheless, the use of graphics to describe the architecture instead of having no description of the architecture at all is an advancement regarding the maintainability of the system.

4.2 Level 2: Modelled

In order to reduce the risk of misinterpretation, this maturity level requires a formal description of the architecture. Architecture Description Languages (ADL) [5] are a possibility to describe an architecture formally, as the semantic is provided by the used ADL. Another kind of formal description is the use of meta-modelling. The semantic is thereby defined within a meta-model which expresses how valid model instances are structured. The probably most popular meta-model to describe software systems is the Unified Modelling Language (UML) [16]. Regarding CBSS, there are further specialized meta-models available which provide aligned modelling capabilities for CBSS. One example of such a component meta-model is the Palladio Component Model (PCM) [18]. In addition to the specified semantics, architecture models also provide the possibility to apply automated analyses (e.g., impact analyzes or performance predictions) or (semi-)automated code generation. However, these are not part of the models but reasoning techniques working upon the models.

The reduction of misinterpretations and the possibility to use automated analyses or code generation techniques are the improvements that are made possible by architectural descriptions of maturity level 2.

4.3 Level 3: Documented

In addition to a level 2 documentation, which focuses on the description of the architecture respectively the structure of the system, this maturity level enjoins to mark design decisions. This means that the use of design patterns (e.g. [6]) and their association to the elements of the architecture have to be indicated. This maturity level demands stating the use of design patterns but does not required to give reasons for the use of the respective design pattern.

Vokác et al. [20] have conducted an controlled experiment in order to identify the influence of documenting the use of architectural design patterns on the maintainability of a software system. They have shown that this leads to a reduction of the maintenance effort.

4.4 Level 4: Traceable

The results of Vokác et al.'s [20] experiment also demonstrated that the use of inappropriate design patterns negatively influence the maintainability. In order to reduce this risk, level 4 stipulates that the already marked design decisions have to be associated with the requirements on the software system, which are the cause for the respective design decision. Software architects are thereby forced to consider the appropriateness of the design patterns and decisions can be more easily traced and checked by others. Furthermore, if the requirements are changed the dependent design decisions can de identified more easily and noted down for reconsideration.

The main benefit of level 4 is the reduction of using inappropriate design patterns and making inappropriate design decisions.

4.5 Level 5: Reasoned

Based on the traceability introduced with level 4, a level 5 architectural documentation additionally requires reasoning of the design decisions. In addition to the association with the causing requirements the architect has to describe the reasons for making the design decision. It is also necessary to mention considered design alternatives and to argue why they are chosen or not. This reasoning of the design decisions increases the comprehensibility for people that are not involved in the design of the system but have to maintain it. Furthermore, dependencies between design decisions (e.g., some decisions make only sense in combination

with other decisions) are emphasized and their connections is directly visible.

The benefit promised by level 5 is to clarify the reasons that led to the architecture and extend the understandability of the architecture to the inherent design decisions. Thus, software architects responsible for the maintenance of a system not only know the architecture itself but also the reasons why it is like that.

5 Conclusion and Outlook

At the beginning, we introduced a refined definition of maintainability focused on the architecture of a software system. The lack of a common meaning of maintainability on the architectural layer was pointed out. We provided definitions to clarify and consolidate the meaning of maintainability on the architectural level in the software engineering community.

Afterwards, the ADMM and each of its five maturity levels were explained in detail. This included the presentation of the method to evaluate the maturity of a system's architecture documentation and description. The ADMM is built in a way that a growing maturity of the documentation is accompanied by a better maintainability of the system's architecture. The ADMM furthermore supports to target the improvement of the documentation as the maturity levels can also be used to identify rewarding aspects of the architectural documentation. In doing so, the ADMM advances the assessment methods for software architectures and thereby eases the assessment for software architects.

The ADMM is the first step towards the AM^3, which will provide an comprehensive evaluation method of an architecture's maintainability. As a next step, we plan to use the ADMM within several industrial projects. The experiences gained within these projects should then be used to validate the ADMM and refine the different maturity levels. We additionally plan to analyze other aspects of a system's architecture which influence the maintainability. Based on these results, we will develop further assessment methods that cover these aspects with the aim to realize the comprehensive AM^3.

References

[1] A. April, J. H. Hayes, A. Abran, and R. Dumke. Software Maintenance Maturity Model (SMmm): the software maintenance process model: Research Articles. *Journal on Software Maintenance and Evolution*, 17(3):197–223, 2005.

[2] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley ; Bonn, 1999.

[3] S. Becker, H. Koziolek, and R. Reussner. The Palladio Component Model for Model-Driven Performance Prediction. *Journal of Systems and Software*, To appear, 2008.

[4] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet. Architecture-level modifiability analysis (ALMA). *Journal of Systems and Software*, 69(1-2):129–147, 2004.

[5] P. C. Clements. A Survey of Architecture Description Languages. In *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*, page 16, Washington, DC, USA, 1996. IEEE Computer Society.

[6] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, 2003.

[7] P. S. Grover, R. Kumar, and A. Sharma. Few useful considerations for maintaining software components and component-based systems. *SIGSOFT Softw. Eng. Notes*, 32(5):1–5, 2007.

[8] P. Hnetynka, F. Plasil, T. Bures, V. Mencl, and L. Kapova. SOFA 2.0 metamodel. Technical report, Dep. of SW Engineering, Charles University, December 2005.

[9] S. Huang and S. Tilley. Towards a documentation maturity model. In *SIGDOC '03: Proceedings of the 21st annual international conference on Documentation*, pages 93–99, New York, NY, USA, 2003. ACM.

[10] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages –, 1990.

[11] ISO. International Standard - ISO/IEC 9126 - 1 (2001). *International Organization for Standardization*, 2001.

[12] M. Kajko-Mattsson, S. Forssander, and U. Olsson. Corrective maintenance maturity model (CM3): maintainer's education and training. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 610–619, Washington, DC, USA, 2001. IEEE Computer Society.

[13] R. Kazman, M. Klein, and P. Clements. ATAM: Method for Architecture Evaluation. Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, 2000.

[14] R. Kneuper. *CMMI*. dpunkt.verlag, 3. edition, 2007.

[15] N. Mari, M.; Eila. The impact of maintainability on component-based software systems. *Euromicro Conference, 2003. Proceedings. 29th*, pages 25–32, 2003.

[16] OMG. Unified Modeling Language (UML). http://www.uml.org/.

[17] L. Pareto and U. Boquist. A quality model for design documentation in model-centric projects. In *SOQUA '06: Proceedings of the 3rd international workshop on Software quality assurance*, pages 30–37, New York, NY, USA, 2006. ACM.

[18] R. Reussner, S. Becker, J. Happe, H. Koziolek, K. Krogmann, and M. Kuperberg. The Palladio Component Model. Technical report, Chair for Software Design & Quality (SDQ), University of Karlsruhe (TH), Germany, May 2007.

[19] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2 edition, 2002.

[20] M. Vokác, W. Tichy, D. Sjoberg, E. Arisholm, and M. Aldrin. A Controlled Experiment Comparing the Maintainability of Programs with and without Design Patterns - A Replication in a Real Programming Environment. *Empirical Software Engineering*, 9(3):149–195, 2004.