The Architecture Documentation Maturity Model ADM^2

Christoph Rathfelder, Henning Groenda

FZI Forschungszentrum Informatik, Karlsruhe
{rathfelder,groenda}@fzi.de

Abstract: Today, the architectures of software systems are not stable for their whole lifetime but often adapted driven by business needs. Preserving their quality characteristics beyond each of these changes requires deep knowledge of the requirements and the systems themselves. Proper documentation reduces the risk that knowledge is lost and hence is a base for the system's maintenance in the long-run. However, the influence of architectural documentation on the maintainability of software systems is neglected in current quality assessment methods. They are limited to documentation for anticipated change scenarios and do not provide a general assessment approach. In this paper, we propose a maturity model for architecture documentation. It is shaped relative to growing quality preservation maturity and independent of specific technologies or products. It supports the weighting of necessary effort against reducing long-term risks in the maintenance phase. This allows to take product maintainability requirements into account for selecting an appropriate documentation maturity level.

1 Introduction

Most of the available software systems are adapted after their creation driven by changes in the business requirements. For example, if new functionality is needed or systems have to be integrated after a merger. A successful evolution during the maintenance phase requires a sound knowledge about existing requirements and their implementation in the system. Otherwise, the quality characteristics of the system cannot be preserved. Design erosion happens and the chance for critical errors rises.

The loss of knowledge over time or by changes in the maintenance personnel can be addressed by documentation. There are several kinds of documentation for a system. For example fine-grained information at source code level, coarse-grained at architectural level, and system-wide at requirements level. The latter two set the big picture and provide a base to comprehend the structure, behavior, rationales, and design decisions. In real life, the most time-consuming task within maintenance activities is the search for this kind of information [Sou98, DLS07]. A big share of this effort can be traced back to the search for high-level information like rationales and design decisions [KLvV]. Besides the included information, the quality of the documentation is also a key factor in preventing design erosion [PB06]. Overall, the documentation at the architecture level and above is a key factor for maintaining systems efficiently in the long-run.

However, the influence of architectural documentation on the maintainability of software systems is neglected in current quality assessment methods. Existing methods provide the assessment of maintainability for a set of anticipated changes, from a process-based or educational viewpoint, or considering specific technological solutions. They all lack a product and technology independent assessment providing general quality statements.

We propose a maturity model for assessing architecture documentation with respect to maintainability. Its maturity levels are shaped according to a growing ability of quality preservation during maintenance. The requirements and characterization stated for each maturity level should support trade-off decisions between higher comprehensibility in the long-term and the effort of creating the documentation. Furthermore, the advantages of automatic knowledge reasoning and provisioning by using formal kinds of documentation should be reflected. This additionally enables assessments tailored to model-driven software engineering environments in which the usefulness of documentation differs compared to classical ones.

The contribution of this paper is the presentation of the multidimensional Architecture Documentation Maturity Model (ADM^2), which includes 1) the effect of documentation on maintainability attributes, 2) independent evaluation dimensions for the degree of formalization and information depth, and 3) a benefit-oriented characterization of the maturity levels.

This paper is structured as follows: Section 2 gives an overview of related work. Section 3 presents the effect of documentation on the different maintainability quality attributes. Section 4 describes the ADM^2 including both of its evaluation dimensions and all of its overall seven maturity levels. Section 5 discusses the benefits promised by each maturity level. Section 6 presents an outlook on the validation plans of the ADM^2. Section 7 concludes the paper and provides an outlook to future work.

2 Related Work

Work related with ADM^2 can be classified into two main categories. The first one subsumes approaches which focus on the maintainability of software systems. The second subsumes approaches focusing on documentation of software systems. Approaches for both categories are presented in the following.

2.1 Maintainability Assessment

Maintainability approaches can be split into three different categories. The first covers scenario-based approaches which evaluate the maintainability of a software system for selected scenarios. The second covers process-based approaches which reason about maintainability solely based on the maturity of maintenance processes. The third category focuses on effects by education, training and knowledge of maintenance personnel. All are presented in the order of enumeration in the following paragraphs.

Nowadays, scenario-based approaches like the Architecture-Level Modifiability Analysis (ALMA) [BLBvV04] or the Architecture Trade-Off Analysis Method (ATAM) [KKC00] are widely used techniques to evaluate the maintainability of a system. These approaches require the definition of scenarios that represent the anticipated evolution of the system. Especially if the planning period is long it is hardly possible to anticipate all needed adaptations of the system. Hence, The uncertainty of assessment results is quite high. As experts estimate the required effort for their implementation, the results of these evaluations strongly depend on the participating individuals and reproducibility between different expert teams is non distinctive. In contrast, our scenario-independent approach allows maintainability assessments by assessing architecture documentation.

The Software Maintenance Maturity Model (SM^{MM}) [AHAD] developed by April et al. allows the evaluation of maintenance activities and the determination of their maturity. Similar to SEI's CMMI, they assess the maintenance process to draw conclusions about quality attributes of maintained products from the process's maturity. The SM^{MM} is based on the assumption that mature processes lead to maintainable systems. It neither evaluates the product itself nor its documentation. It also provides no assistance on how to increase the maintainability of a system.

An additional maturity model with relation to maintainability is the Corrective Maintenance Maturity Model (CM^3) [KMFO01]. In this model, the capability of an enterprise to maintain systems is considered from an educational point of view. It is focused on the knowledge and training of maintenance engineers and based on the assumption that well-educated engineers produce maintainable systems. Hence, there are no guidelines how the maintainability of a software system can be evaluated.

2.2 Documentation Assessment

The different existing assessment approaches for the quality of a system's documentation are scenario-independent and consider documentation from a generic and broad viewpoint. These approaches are briefly described in the following.

Pareto and Boquist developed a quality model for design documentation based on the results of their quality model survey [PB06]. They identified overall 22 quality attributes for 6 different quality characteristics which effect the quality of design documentations. They regard design documentation as documentation on artifacts on the abstraction level between requirements specifications and code. They identify the 22 attributes but no metrics or guidelines about their characteristic maturity benefits are pointed out. Trade-off decisions are therefore not supported. The authors concentrate on documentation in model-centric projects. Hence, the identified different quality characteristics are regarded purely from a documentation data handling perspective. This renders reasoning about the effects of documentation, for example on maintainability, cumbersome.

Huang and Tilley described their idea of a Documentation Maturity Model (DMM) in [HT03]. The DMM has five maturity levels and is focused on the perception of documentation by software engineers in terms of ease of interpretation. Each level requires a different set of presentation techniques, for example level 3 requires animated graphics and hyperlinks. They do not consider the information contained in the documentation. There is no distinction between the different application levels of documentation, for example code, architecture, or requirements level. By focusing on human interpretation, their approach is not reflecting how pay-off for formalized documentation differs with respect to different kinds of documentation, for example in model-driven software engineering environments.

3 Effects of Documentation on Maintainability

Understanding the effect of documentation on maintainability first requires a thorough definition of maintainability. There is a number of different maintainability definitions available, for example as discussed in [BDP]. Some approaches view maintainability purely at the code level, whereas other reflect specific issues at the architecture level. For example Grover et al. [GKS07] take into account that on the architecture level a reconfiguration or arrangement of components and their interconnection is more likely than a complete restructuring and recoding. Our focus is on the effect of architecture documentation on maintainability. We clarify and introduce our refined view on maintainability in this section and point out the extent of effects on maintainability attributes caused by documentation.

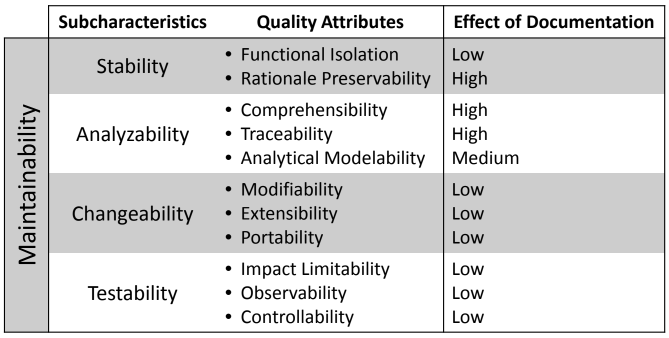
Our definition of maintainability is based on the ISO/IEC 9126 standard [ISO01] which provides a complete quality model. The model covers the characteristics functionality, reliability, usability, efficiency, maintainability, and portability. The standard describes maintainability as "the capability of the software to be modified" and provides the subcharacteristics shown in Figure 1. In addition to this quality definition we use the term architecture as provided by Bass et al. [BCK99]:

> "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." [BCK99]

	Subcharacteristics	**Capability Description**
Maintainability	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 a software system to be tested effectively in order to observe and check its behavior.

Figure 1: Meaning of Maintainability according to ISO/IEC 9126 [ISO01]

Our view on maintainability is illustrated in Figure 2. We divided each subcharacteristic further into several quality attributes which represent independent viewpoints on that subcharacteristic. These attributes and how strongly they are affected by documentation is pointed out in the following paragraphs.



	Subcharacteristics	Quality Attributes	Effect of Documentation
Maintainability	Stability	• Functional Isolation	Low
		• Rationale Preservability	High
	Analyzability	• Comprehensibility	High
		• Traceability	High
		• Analytical Modelability	Medium
	Changeability	• Modifiability	Low
		• Extensibility	Low
		• Portability	Low
	Testability	• Impact Limitability	Low
		• Observability	Low
		• Controllability	Low

Figure 2: Effect of Documentation on Maintainability

Stability. Stability is the capability to avoid that modifications cause unexpected effects on other parts of the software system. It is based on the structure of a system and depends on the two aspects functional isolation and rationale preservability. *Functional Isolation* addresses the grouping and isolation of different functionality within a system. If functions are changed or replaced, a strong grouping reduces unexpected effects in unchanged groups. However, extra-functional quality attributes (e.g., performance or availability) of the system and other components may still be affected by such changes. Documentation should be used to describe function groups, but it has no effect on the separation itself. The effect is therefore rated low. *Rationale Preservability* addresses the continuity of areas in the designed system architecture. Areas within an architecture have a specified semantic and are usually defined according to architectural patterns. They can be seen as design rationales. For example, the use of the Model-View-Controller pattern leads to 3 areas for models, views and controllers. These areas should be preserved over the lifetime. Changes on the system only lead to component assignment or removal from the areas. Documentation of rationales has a high impact on the continuity of the design. It states important design issues at a high level and is a key to prevent design erosion.

Analyzability. Analyzability is the capability needed to search out deficiencies and causes of failures within the system. It can be broken down into the three aspects of comprehensibility, traceability, and analytical modelability. It can be seen either from a human or machine centric perspective. *Comprehensibility* addresses how easily engineers can understand the system and its architecture. Documentation is responsible for providing detailed and high-level information. Hence, it has a high influence on comprehensibility. *Traceability* addresses how requirements, design rationales, decisions, and even discarded alternatives are linked. Tracing back decisions to rationales or requirements allows re-examining them at any time. Especially the links to discarded alternatives attached with reasons or measurements may be valuable if the made decision is revisited at a later point in time. Hence, documentation has a high effect on traceability. *Analytical Modelability* addresses how easy models for automated analyses can be built for the system. There are tools and analyses which can extract necessary information from design models, source-, or object code but in most cases additional knowledge of engineers is necessary. Storing this additional information in the documentation fosters its reuse. Examples for automated analyses are the component interaction checker SOFA [HPB$^+$05] and the performance prediction approach Palladio [Bec]. Overall, documentation has a medium effect on analytical modelability.

Changeability. Changeability is the capability to extend, enhance, and customize a software system. It can be broken down into the three aspects modifiability, extensibility, and portability as Matinlassi and Nimelä have shown in [MN03]. It focuses on the changes or adaptations themselves. *Modifiability* addresses how the system can be restructured in order to meet new or changed requirements (e.g. a shorter response time). Documentation of the traceability can ease a design preserving restructuring and support quality assurance by analytic checks of requirements (e.g. runtime constraints). However, documentation has only a low effect on the modifications themselves. *Extensibility* addresses how the system can be extended with new functionality or function groups. Documentation of explicitly provided extension points can ease this kind of changes, but the structural aspect of extension points being there in the first place is definitely bigger. Hence, documentation has a low effect on extensibility. *Portability* addresses how the system can be adapted to other environments (e.g. another operating system or middleware). Documentation for the subcharacteristic stability can ease portability (e.g. if abstraction layers are used), but its general effect on portability is low.

Testability. Testability is the ability of a software system to be tested effectively in order to observe and check its behavior. It can be broken down into the three aspects of impact limitability, observability, and controllability. The two latter ones are already discussed in detail in [Bin94] and are important properties in unit testing. Documentation has in general only low effect on testability although the documentation of design patterns for example may increase the testability of an architecture [CKvS05]. *Impact Limitability* addresses how changes and their effects can be restricted to parts of the system. If limitability is high, it is sufficient to test the restricted parts. *Controllability* addresses how fine-grained the state of the system and its components is controllable when its behavior is examined. *Observability* addresses how fine-grained the state of a system and its components can be observed from the outside.

4 Architecture Documentation Maturity Model

In this section, we describe the developed Architecture Documentation Maturity Model (ADM^2). Based on the refined definition of architectural maintainability presented in the previous section, the ADM^2 maturity levels allow an assessment of an architecture's documentation with respect to maintainability of the system. The development of the ADM^2 is based on a sound literature review as well as the experiences we have gained within different industrial and research projects First, we introduce the two dimensions that are used to evaluate maturity. Second, we show the different maturity levels of each evaluation dimension including a description of the characteristic attributes of an architecture's documentation on the respective maturity level.

4.1 Evaluation Dimensions

In [RG08], we have sketched our ideas of a one-dimensional documentation maturity model. We validated that model in industrial projects and in discussions with several software architects. The outcome showed that a one-dimensional maturity model is not sufficient to assess maturity with respect to maintainability. A differentiation between the information included in the documentation and the formal techniques used for documenting this information is necessary. These two dimensions are independent of each other, which mean that the maturity in one dimension does not influence the maturity in the other dimension.

The first evaluation dimension is called *Information Depth* and regards the information and knowledge included in the documentation. The importance of information depth for the maintainability is for example investigated in the study conducted by Forward and Lethbridge [FL02]. The study shows that in their case the content of documentation has a larger influence on maintainability than the formal techniques used.

The second evaluation dimension is called *Formalization* and focuses on the formalization degree of the documentation. The use of formalized models not only reduces the risk of misinterpretation but also enables automated processing of information. For example, Grisham et al. [GHP07] showed that formalization promises an increasing quality of the design decisions made. The type of representation of the documentation plays only a minor role. For example, the study of Forward and Lethbridge [FL02] shows that there is no clear distinction if using graphical or textual representation for documentation is an advantage. Grönniger et al. [GKR$^+$] pointed out areas in which textual representations are more efficient although engineers often regard graphical representations as catchier. The ADM^2 does not differentiate between a textual or graphical representation of the architecture.

Figure 3 visualizes the two evaluation dimensions and sketches the maturity levels. We describe the 7 different maturity levels of the ADM^2 for each dimension in detail in the following subsection.

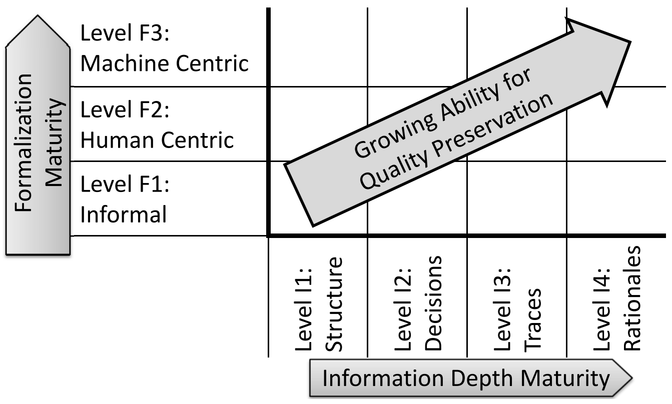


Figure 3: ADM^2 Overview

4.2 Maturity Levels

The existence of an architectural documentation is a precondition for the ADM^2, hence systems without any architectural documentation cannot be ranked in the ADM^2. However, they can be subsumed as a virtual level 0 to point out their immaturity. The maturity levels are arranged in an ascending order for each of the two dimensions. The documentation characteristics required on a maturity level include all characteristics mandatory on lower levels of that dimension. Growing maturity of the documentation thereby goes along with an increased maintainability. Lifting the maturity of a system to a higher level initially induces effort. However, the increased maintainability could compensate these costs over the life-time of the system depending on the system and situation. Evaluators of documentation maturity should take this into account when selecting the appropriate maturity level for a system.

The description of the maturity levels within this section starts with the formalization dimension which is followed by information depth's dimension.

4.2.1 Formalization Maturity

A higher maturity of the documentation in this dimension is accompanied by the use of more formal models. These provide a fixed semantic meaning of the modeled architectural artifacts and thereby ease comprehension of the system. Furthermore, the use of formal models eases (semi-)automated analyses which can for example be used to estimate the impacts caused by architectural modifications. Model-driven techniques can be applied to ensure a consistent documentation by automatically propagating changes in the architecture to the implementation and vice versa. In the following, we describe the three maturity levels of this dimension, namely *(F1) Informal*, *(F2) Human Centric*, and *(F3) Machine Centric*, in more detail.

Level F1: Informal. On this maturity level, the documentation consists of simple graphics or textual descriptions. The architecture documentation has no predefined semantics. For example, tools like Microsoft PowerPoint or Word provide the respective drawing and writing capabilities. In the case of a textual representation, a common glossary which defines the meaning of the used terms is missing on this maturity level.

Level F2: Human Centric. In order to reduce the risk of misinterpretation, this maturity level requires a semantic description within the architecture's documentation. Architecture documentations which are rated on this level are mainly used by humans, as the formalization of the documentation is too low to allow automated processing of the models. There are several possibilities to add semantic description to the documentation. In a textual representation, a common glossary can be used, to define the meaning of different terms. In a graphical representation, the use of a common set of symbols each having a defined semantic is an adequate solution. The semantic of the symbols can be specified individually in a legend or a common standardized set of predefined symbols can be used. The probably most popular set of symbols to describe software systems is the Unified Modeling

Language (UML) [OMG]. The shortcomings of UML and why it can not be considered as fully formalized language are pointed out in [HS05].

Level F3: Machine Centric. The documentation on this maturity level must follow a strict grammar or meta-model to be machine readable and processable. This requires a much stricter semantic definition compared to level F2. Architecture Description Languages (ADL) [Cle96] can be used for this kind of formal description. The languages or meta-models used on this level are often specialized to a certain domain (e.g. embedded controllers) and therefore also known as Domain Specific Languages (DSL). Regarding component-based software systems for example, there are specialized meta-models available which provide aligned modeling capabilities.

4.2.2 Information Depth

As already depicted in Figure 3 the Information Depth dimension is split into the four maturity levels *(I1) Structure*, *(I2) Decisions*, *(I3) Traces*, and *(I4) Rationales*. They are explained in the following.

Level I1: Structure. This first maturity level requires the existence of an architectural documentation that includes a description of the system's structure. This description must include the connections and dependencies between different components of the system and should give an overview on the structure of the system.

Level I2: Decisions. In addition to I1, this maturity level stipulates to mark design decisions. This means for example that the use of design patterns (e.g. [Fow03]) and their association to the elements of the architecture have to be indicated. Making decisions is an essential part of an architecture's development [TA]. An explicit marking of the design decisions prevents architects to repeat a design decision several times. In [ZG] and [CND07], two solutions that support a formalized modeling of design decisions are proposed.

Level I3: Traces. This level 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. The results of Vokác et al.'s [VTS$^+$04] experiment demonstrate that the use of inappropriate design patterns negatively influence the maintainability. Because of the explicit linking of design decisions and requirements, software architects are forced to examine the appropriateness of the design patterns and their decisions more clearly. Based on the UML-profile proposed by Zhu and Gorton [ZG], it is possible to formally specify requirements.

Level I4: Rationales. Based on the traces introduced with the level I3, a I4 architectural documentation requires reasoning on design decisions. In addition to the association with the respective 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. Furthermore, dependencies between design decisions (e.g., some decisions make only sense in combination with other decisions) are emphasized and their connections are directly visible. Reasoning on the design decisions and their rationales increases the comprehensibility. In [KLvV], Kruchten et al. present an ontology that supports an automatic reasoning on design decisions. However, they also mention, that documenting design rationales is still only seldom used in practice.

5 Application Benefits of the ADM^2

In this section we point out the benefits regarding the system's maintainability that come along with each maturity level. The ISO/IEC 12207 [IEE08] defines a common framework for software life cycle processes and describes software maintenance as one of the primary processes in the life cycle of a software product. Based on these general life cycle processes, the ISO/IEC 14764 standard [IEE06] focuses on maintenance activities and defines the software maintenance cycle. This cycle consists of the three phases *Problem and Modification Analysis*, *Modification Implementation*, and *Maintenance Review / Acceptance*. In Figure 4 we sketch this maintenance cycle, whereas we adjusted the naming of the phases a little bit to clarify their content.

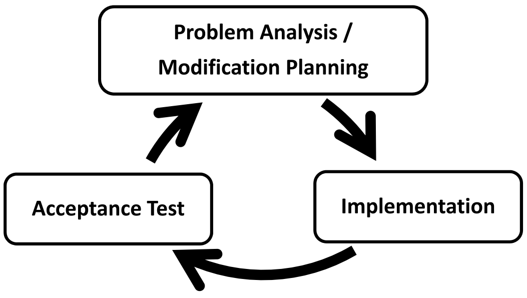


Figure 4: Software Maintenance Cycle [IEE06]

Based on this refined view on maintenance, we describe the influences of documentation on the different maintenance phases. For each maturity level, we discuss in which way the mandatory characteristics lead to a reduction of the maintenance effort in these phases. We firstly focus on the dimension of Formalization and afterwards on the Information Depth dimension.

5.1 Formalization Maturity

Level F1: Informal. The use of an informal architectural documentation eases the comprehension of the architecture and can be used to gain an abstract overview of the architecture. As shown in [FL02], this is an essential part during the *Problem Analysis / Modification Planning* phase. Nevertheless, the lack of semantics on the first level might lead to misinterpretations of the graphics and descriptions, 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, the probability of producing inconsistent views within the documentation is quite high.

Level F2: Human Centric. The reduction of misinterpretations is the main benefit of this level. This is achieved, as architects are forced, to use a common language to describe the architecture and the included information. Due to the predefined set of symbols or terms, it is possible to use specialized tools. These tools provide maintenance engineers a better support for their activities than the general tools like MS Word for example. For these reasons, a documentation of the second maturity level may increase the efficiency in the *Problem Analysis / Modification Planning* phase. The improved tool support in combination with semi-formal defined description languages also promises an increasing consistency of implementation and documentation and thereby improves the efficiency of the *Implementation* phase.

Level F3: Machine Centric. In addition to an accurate semantical description, the possibility to apply automated analyses (e.g., impact analyses or performance predictions) is an improvement promised by this maturity level. The automated analyses further improve the efficiency within the *Problem Analysis / Modification Planning* as they support the investigation of problem, for example analyzing performance bottlenecks [BDIS04]. Additionally, they can also be applied to evaluate and weigh different design alternatives. This improves the quality of the planed modifications. Furthermore, the formalized documentation can be used to reason about the included architectural knowledge as envisioned by Kruchten et al. [KLvV]. The formalized architectural model also improves the *Implementation* phase is as they enable the automated generation of code using model-driven techniques. Moreover, the consistency of model and code is increased due to the automated transformations. If these transformations are coupled with the automated analyses, the quality of the results can be further improve as shown in [Bec]. In the *Acceptance Test* phase, the formalized models likewise form the bases for an automation of some activities. As shown by Grundy et al. in [GCL04], it is possible to generate test cases based on a formalized model. In this way, the effort in the *Acceptance Test* phase can be reduced.

5.2 Information Depth Maturity

In the following paragraphs we focus on the improvements regarding the maintenance effort that are proposed by a growing maturity of the knowledge included in the documentation. These improvements are mainly concentrated in the *Problem Analysis / Modification Planning* phase, whereas there are also some smaller effects on the *Implementation* and *Acceptance Test* phase.

Level I1: Structure. The explicit documentation of the system's structure is the main benefit of this level. As shown in [FL02], the extraction of a system's overview and the detection of relations between different components is an important activity for maintenance engineers. An explicit documentation of the structure and relations reduces the effort for this activity, as the engineers do not any more need to extract the structure the source code. If formalized models are used which are rated on level F3 the advantages of automated analyses and prediction as already shown above can be reached even on this first maturity level.

Level I2: Decisions. The documentation of design decisions required on this level further increases the comprehensibility. It also promises to impede the repetition of design decisions during the lifetime of a software system [CND07]. Maintenance engineers often have to capture design decisions to plan modifications. As shown [ZG], the documentation of the structure only does not aid them. For this reason, the explicit documentation of design decision supports maintenance engineers in their work. The experiments conducted by Vokác et al. [VTS$^+$04] and Prechelt et al. [PULPT02] show that documenting the use of design patterns reduces the maintenance effort in the *Problem Analysis / Modification Planning* and *Implementation* phase.

Level I3: Traces. The documentation of the associations between design decisions and causal requirements is a central characteristic of level I3. Changes of requirements often lead to changes of some design decisions. The explicit linking of decisions and requirements eases the location of design decisions that have to be reconsidered when requirements are changed. This leads to a reduction of the effort required for the *Problem Analysis*. The study described in [dSAdO05] also emphasizes that requirements are an important part of an architecture's documentation used for maintenance. The link to requirements forces architects to think over their design decisions, which might lead to a reduction of using inappropriate design patterns. Thus, the quality of the results of the *Modification Planning* activities is improved. The linking of requirements and design decisions also influences the *Acceptance Test* phase, as the traces ease the detection of requirements that might be affected by changes within the architecture and therefore need to be tested again.

Level I4: Rationales. The explicit documentation of design rationales clarifies the reasons that led to the architecture and thus improves the comprehensibility of the architecture and the inherent design decisions [GHP07]. Maintenance engineers thereby do not only know the architecture itself but also the reasons why it is like that. Software architects and maintenance engineers are forced to argue their rationales. Thus, the probability to make inconsistent design decisions is reduced and the quality of the planned modifications as increased. Additionally, Zimmermann et al. [ZGK$^+$] show that explicitly modeled design decisions and their rationales promise the benefit of being reused within different software projects. Although this sparsely influences the maintainability of a single system, the reuse of design decision increases the efficiency if different systems are developed or maintained.

6 Validation

The validation of a maturity model is an extensive task. We initially performed a validation of the applicability of the ADM^2 by evaluating different architectural documentations using the ADM^2. As application of maturity models cannot lead to antithesis and rejection of the model, we strive for a validation of the benefits proposed by each level. We plan to perform this kind of validation by an empirical case study with a group of computer science students from Universität Karlsruhe (TH) as well as discussions with experienced maintenance engineers and the application of the ADM^2 in industrial projects. As a lot of these projects at our research institute are focused on restructuring of evolved software systems, we expect to substantiate the results of the experiment.

The empirical experiment used to validate the benefits and the shaping of the models is set up as follows. All participating students are split up into different groups and receive an identical maintenance task for a software system. However, the provided architecture documentation is different for each group. It is derived from a mature documentation by removing parts which are mandatory on a higher level (e.g., the design rationales or the traces). We also vary the formalization level of the documentation. For example by removing the glossary and substituting DSL by UML symbols and UML symbols by other symbols like boxes that have no predefined meaning. The time required for completing the task is captured for each phase of the maintenance cycle (see Figure 4) separately. Based on these measurements, we expect to be able to show that increased documentation maturity provides the proposed benefits and that the levels are shaped according to ascending maintainability.

7 Conclusion and Outlook

In this paper, we first presented and discussed the effects of documentation on different maintainability quality attributes. Second, we introduced the Architecture Documentation Maturity Model (ADM^2) and showed the necessity of two independent different evaluation dimensions for Formalization and Information Depth. Third, we described for each of the overall 7 maturity levels their shaping as well as their characteristics and additionally provided tool examples. Fourth, we presented the maintainability benefits gained for each maturity level and their share in the different phases of the maintenance cycle. Fifth and last, we elaborated on the current and planned validation of the ADM^2.

The proposed maturity model aids software engineers to evaluate the maturity of existing documentation and plan documentation for new projects. As it is designed in a way that a growing maturity of the documentation is accompanied by a better maintainability, identifying the appropriate maturity levels is much easier compared to other approaches. Planning appropriate documentation with the ADM^2 even supports to select different maturity levels for different partitions of a project. In contrast to other approaches, this tailoring enables fine-granular trade-off decisions, for example to take areas into account where model-driven software engineering techniques are used or not used. The ADM^2 furthermore differentiates itself by supporting targeted improvements of the documentation as the maturity levels can also be used to identify rewarding aspects of the architectural documentation.

As we pointed out in section 6, a more detailed refinement and validation of the ADM^2 is planned, consisting of its application in industrial projects, discussions with experienced maintenance engineers and researches, and an empirical study. In the future, we also want to examine other areas than documentation to build a general Architecture Maintainability Maturity Model.

References

[AHAD] Alain April, Jane Huffman Hayes, Alain Abran, and Reiner Dumke. Software Maintenance Maturity Model (SMmm): the software maintenance process model: Research Articles. *Journal on Software Maintenance and Evolution*, 17(3):197–223.

[BCK99] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley ; Bonn, 1999.

[BDIS04] Simonetta Balsamo, Antinisca Di Marco, Paola Inverardi, and Marta Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE Transactions on Software Engineering*, 30(5):295–310, May 2004.

[BDP] Manfred Broy, Florian Deissenboeck, and Markus Pizka. Demystifying Maintainability. In *Proc of the WoSQ '06*.

[Bec] Steffen Becker. Coupled Model Transformations. In *WOSP '08*.

[Bin94] Robert V. Binder. Design for testability in object-oriented systems. *Commun. ACM*, 37(9):87–101, 1994.

[BLBvV04] PerOlof Bengtsson, Nico Lassing, Jan Bosch, and Hans van Vliet. Architecture-level modifiability analysis (ALMA). *Jour. of Syst. and Softw.*, 69(1-2):129–147, 2004.

[CKvS05] Roberta Coelho, Uirá Kulesza, and Arndt von Staa. Improving architecture testability with patterns. In *in OOPSLA '05*, 2005.

[Cle96] Paul C. Clements. A Survey of Architecture Description Languages. In *Proc. of IWSSD '96*. IEEE, 1996.

[CND07] Rafael Capilla, Francisco Nava, and Juan C. Duenas. Modeling and Documenting the Evolution of Architectural Design Decisions. In *Proc. of SHARK-ADI '07*. IEEE, 2007.

[DLS07] Sumita Das, Wayne G. Lutters, and Carolyn B. Seaman. Understanding documentation value in software maintenance. In *Proceedings of CHIMIT '07*. ACM, 2007.

[dSAdO05] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *Proc. of SIGDOC '05*, pages 68–75. ACM, 2005.

[FL02] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proc. of DocEng '02*, 2002.

[Fow03] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, 2003.

[GCL04] John Grundy, Yuhong Cai, and Anna Liu. SoftArch/MTE: Generating Distributed System Test-Beds from High-Level Software Architecture Descriptions. In *Automated Software Engineering*, volume 12, pages 5–39, December 2004.

[GHP07] Paul S. Grisham, Matthew J. Hawthorne, and Dewayne E. Perry. Architecture and Design Intent: An Experience Report. In *Proc. of SHARK-ADI '07*. IEEE, 2007.

[GKR$^+$] H. Grönniger, H. Krahn, B. Rumpe, M. Schindler, and S. Völkel. Textbased Modeling. In *in ATEM '07*.

[GKS07] P. S. Grover, Rajesh Kumar, and Arun Sharma. Few useful considerations for maintaining software components and component-based systems. *SIGSOFT Softw. Eng. Notes*, 32(5):1–5, 2007.

[HPB$^+$05] Petr Hnetynka, Frantisek Plasil, Tomas Bures, Vladimir Mencl, and Lucia Kapova. SOFA 2.0 metamodel. Technical report, Dep. of SW Engineering, Charles University, December 2005.

[HS05] Brian Henderson-Sellers. UML - the Good, the Bad or the Ugly? Perspectives from a panel of experts. *Software and System Modeling*, 4(1):4–13, 2005.

[HT03] Shihong Huang and Scott Tilley. Towards a documentation maturity model. In *Proc. of SIGDOC '03*, pages 93–99, 2003.

[IEE06] IEEE. International Standard - ISO/IEC 14764 IEEE Std 14764-2006. 2006.

[IEE08] IEEE. Systems and Software Engineering - Software Life Cycle Processes. *IEEE STD 12207-2008*, 2008.

[ISO01] ISO. International Standard - ISO/IEC 9126 - 1 (2001). *International Organization for Standardization*, 2001.

[KKC00] Rick Kazman, Mark Klein, and Paul Clements. ATAM: Method for Architecture Evaluation. Technical Report CMU/SEI-2000-TR-004, SEI, 2000.

[KLvV] Philippe Kruchten, Patricia Lago, and Hans van Vliet. Building Up and Reasoning About Architectural Knowledge. In *QoSA 06*.

[KMFO01] Mira Kajko-Mattsson, Stefan Forssander, and Ulf Olsson. Corrective maintenance maturity model (CM3): maintainer's education and training. In *Proc. of ICSE '01*, 2001.

[MN03] Mari Matinlassi and Eila Niemela. The impact of maintainability on component-based software systems. *Proc. of Euromicro '03.*, pages 25–32, 2003.

[OMG] OMG. Unified Modeling Language (UML).

[PB06] Lars Pareto and Urban Boquist. A quality model for design documentation in model-centric projects. In *Proceedings of SOQUA '06*, pages 30–37. ACM, 2006.

[PULPT02] Lutz Prechelt, Barbara Unger-Lamprecht, Michael Philippsen, and Walter F. Tichy. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE TSE*, 28(6):595–606, 2002.

[RG08] Christoph Rathfelder and Henning Groenda. Towards an Architecture Maintainability Maturity Model (AM3). *Softwaretechnik-Trends*, 28(4):3–7, November 2008.

[Sou98] Maria Joao Sousa. A Survey on the Software Maintenance Process. In *Proceedings of ICSM '98*, page 265. IEEE, 1998.

[TA] Jeff Tyree and Art Akerman. Architecture Decisions: Demystifying Architecture. *IEEE Softw.*, 22(2):19–27.

[VTS$^+$04] Marek Vokác, Walter Tichy, Dag Sjoberg, Erik Arisholm, and Magne 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.

[ZG] Liming Zhu and Ian Gorton. UML Profiles for Design Decisions and Non-Functional Requirements. In *SHARK-ADI '07.*

[ZGK$^+$] Olaf Zimmermann, Thomas Gschwind, Jochen Malte Küster, Frank Leymann, and Nelly Schuster. Reusable Architectural Decision Models for Enterprise Application Development. In *QoSA07.*