Towards the Essentials of Architecture Documentation for Avoiding Architecture Erosion

Sebastian Gerdes Stefanie Jasser Matthias Riebisch

Sandra Schröder Mohamed Soliman Tilmann Stehle

Department of Informatics, University of Hamburg
Hamburg, Germany
{gerdes, jasser, riebisch, schroeder, soliman, stehle}@informatik.uni-hamburg.de

ABSTRACT

Software architecture documentation is essential for preventing architecture erosion that is a major concern of sustainable software systems. However, the high effort for elaboration and maintenance of architecture documentation hinders its acceptance in practice. Most state-of-the-art research methods assume comprehensive architecture documentation. By reducing architecture documentation to those aspects that are most important for architecture erosion, we want to achieve more acceptance for architecture documentation especially in agile projects. This reduction, however, has effects on architecture-related activities during software design and implementation.

Keywords

Software architectures, Agile software development, Architecture description languages, Software evolution

1. INTRODUCTION

Software architectures are widely accepted as crucial for developing complex and long-living software systems. Architecture documentation is frequently considered as a potential overhead effort. This holds especially true for agile development processes which are considered mainstream today. On the other hand, documenting software architecture and the underlying decisions is essential to communicate with developers, in order to make them accept and understand the proposed architecture. This activity is known as *Architecture Enforcement* [12]. Supporting developers to have a better understanding and acceptance for the proposed architecture would result in a more disciplined implementation of the architecture, which would consequently reduce the possibility of architecture erosion [7] as one factor. Most works in academic research on architecture sustainability assume fully documented architectures, as for example the methods on impact analysis based on traceability. A high effort for architecture documentation would be required to transfer these methods into industrial practice. In addition, efforts in software architecture documentation (see Section 2) do not provide enough guidance to support developers perceiving it.

With this position and vision paper, we present research efforts extending a recent study [8] towards identifying aspects that are essential for architecture documentation during evolution and which address the actual needs of software architects and software developers who we consider as the main stakeholders of an architecture documentation. The vision behind our research is twofold: an increased acceptance for architecture documentation in practice by reducing the effort for establishing and maintaining it, as well as supporting developers to understand and implement software architectures correctly.

2. RELATED WORK

Documentation of software architectures can be accomplished in various ways:

Informal documentation like drawings and associated text are still widely used in industry [2]. In general, this kind of documentation is unstructured, ambiguous and hardly maintainable especially when systems evolve.

Semi-formal documentation consist of syntactically defined elements with informally described semantics, such as the de-facto industry standard Unified Modeling Language.

Formal documentation with formally defined syntax and semantics are represented for example by several formal Architecture Description Languages (ADL). Their formal and detailed specification allows for tool-supported conformance and consistency checking. However, a recent survey [5] revealed that the majority of practitioners either stopped using formal ADLs or did not even consider to use them, because they considered ADLs as too heavy-weight.

Documentation of decisions. In the past decade there was a paradigm shift towards documenting design decisions [4]. One of the first means to document them are design decision templates [10] followed by various approaches in the field of architecture knowledge management [9].

In agile approaches of software development, effort reduction and simplification led to the goal that software architecture documentation should only contain critical aspects,

SAGRA'16, Nov 28 - Dec 2, 2016, Copenhagen, Denmark.

DOI:

such as architecturally significant requirements or diagrams showing critical views [1]. Hadar et al. proposed the Abstract Architecture Specification document containing the relevant and updated information [3].

Prioritization occurs for example in requirements engineering as well as in architecture design methodologies. Utility trees represent an example, established for describing hierarchies of goal refinement and for expressing priorities of goals, for example as part of the Attribute-Driven Design methodology ADD [11].

3. ENFORCEMENT CONCERNS TO REDUCE ARCHITECTURE EROSION

This paper focuses on architecture enforcement. The aim of this process is twofold. Firstly, it means sharing the results with stakeholders - especially developers - and getting them accepted [12]. Moreover, this process also encompasses architecture conformance checking, which means to assure that decisions are implemented as intended by the architect, in order to minimize architecture erosion. Regardless of the individuality of development projects, in [8] we identified concerns that are generally considered important by software architects during architecture enforcement and to reduce architecture erosion. In this study, we interviewed 12 experienced software architects from industry. In the following, we present some of our findings briefly:

for example, we found out that architects differentiate between *macro and micro architecture*. Those two views refer to the level of architecture detail. The macro architecture represents the general idea of the system and its fundamental architecture decisions, e.g. on structures, components, data stores or architecture styles. The micro architecture refers to the architecture within a specific component and its detailed design. The micro architecture can be considered as the responsibility of a skilled developer and does not have to be documented in the minimized architecture documentation. Architects should concentrate on documenting the macro architecture.

Another interesting concern mentioned by experts was *appropriate use of technology*. As shown in the survey of Miesbauer et al., most of the architecture decisions are technology decisions [6], e.g. concerning frameworks, programming languages or platforms. In our study, experts emphasized that it is crucial to monitor how a specific technology is used by developers. Technologies offer complex functionality. Architectural rules can be easily violated if technologies are not used in the intended way. That is why it could be helpful to document how a chosen technology is supposed to be used in the development project.

Patterns are also an important concern. Patterns can be applied on different abstraction levels, from architecture, to design and implementation. While the architect is considered to be responsible for patterns on design level, patterns on implementation level are at the developers' discretion. In order to effectively guide the implementation (demand D3, see next Section) and assess the architecture (demand D4), the software architecture documentation should be able to record the most important constraints regarding those patterns. Even better, architecture patterns and styles and the corresponding constraints should be expressible in testable rules.

Other concerns mentioned by experts encompass architecture principles, design for testability and visibility of domain concepts, just to name a few. The full list of identified concerns and corresponding explanations are given in [8].

Based on our findings we identified the need for a software architecture documentation that helps architects - and developers - focusing on the most important concerns during architecture enforcement, thus reducing architecture erosion as one important factor. In the next sections we present the demands to be fulfilled by such a documentation (Section 4). Moreover, we propose a process (Figure 2) which helps deciding which concerns should be documented and which of them should be documented semi-formally or even in a formal way.

4. DEMANDS TO BE FULFILLED BY ARCHITECTURE DOCUMENTATION

In this section, we present four demands for an architecture documentation D1 ... D4 in order to use it effectively for architecture enforcement and as one factor preventing architecture erosion during evolution:

D1: Preserving architecture knowledge. The minimized architecture documentation needs to support the prevention of uncontrolled loss of architecture knowledge for effective maintenance and evolution of software architecture.

D2: Facilitating Communication. The minimized architecture documentation needs to facilitate the communication between architects and developers, for example through the definition of a vocabulary to reason about a software system's essentials. Further support is provided through the improved comprehension, which allows focusing on discussed elements. Moreover, the documentation should strive for clarity and a shared understanding between stakeholders, especially software architects and software developers.

D3: Guiding the implementation. The minimized documentation will guide the implementation and changes effectively by providing the information needed by developers and maintainers and encourages a good comprehension of the software architecture for those stakeholders. In order to achieve this, the documentation should provide enough information for them so that they are able to correctly implement architecture decisions and additionally recognize if their implementation adheres to the intended architecture and the corresponding architecture rules.

D4: Support for Architecture Assessment. The minimized documentation helps to validate and assess the architecture in terms of architecture conformance checking, i. e. comparing the implemented architecture with the intended architecture. In this way, it helps the architect to monitor and control architecture evolution, in order to prevent architecture erosion and degraded software quality. That is why the architecture documentation must record the architecture rules that have to be respected by the implementation. It should be possible to document which kind of violations can possibly occur during implementation. Having this information helps the architect to focus on the most risky parts of an implementation during a code review. Furthermore, the documentation needs to define how much flexibility is allowed for developers, i.e. when they are allowed to break certain rules and what aspects concerning architecture must be definitely followed. In order to do this, it is required that those aspects are appropriately formalized. In Section 5, we will describe how to decide under which conditions an archi-

tecture solution should be formalized and when no documentation is necessary, or a semi-formal documentation suffices.

5. PRIORITIZING ARCHITECTURAL ASPECTS

In order to minimize the effort which is needed to document architecture decisions, the architecture's documentation has to be reduced to its essentials. Although identifying the architecture's essentials induces additional efforts, it helps to produce a useful architecture documentation that developers can easily perceive. The extent of documentation effort should represent the importance of the documented architecture solution. This section introduces a process that helps an architect to identify architecture essentials and decide on the appropriate level of documentation for a planned architecture solution.

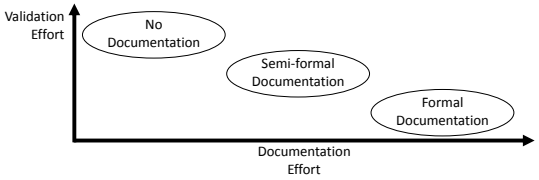

Figure 1: Three Levels of Documentation

We distinguish three levels of documentation as depicted in Figure 1: the architect can decide not to document a solution at all. His second option is to document it semi-formally. Semi-formal means of expression have a well-defined syntax. Their semantics, however, are defined ambiguously using natural language. Furthermore, they sometimes define syntax elements for additional undefined extensions to the basic syntax and semantics. An example of a semi-formal document is a UML-diagram using notes and project-specific stereotypes. Thirdly, the architect can document his decisions using a formal language. The three options differ in the extend of effort necessary to apply them (documentation effort) and their utility for validating implementations against the planned architecture solution (saved validation effort). Furthermore, the amount of details represents another way to decide on the documentation effort, even if not considered here.

We propose a process for deciding on the appropriate level of documentation as depicted in Figure 2. Obviously, the decision on what is essential depends on the goals of the project. As a running example, we assume a project with the goal to provide a cloud service for applying filters to images.

The first step of the process is to define the Non-Functional Requirements (NFRs) as clearly as possible. This is done in cooperation with the customer, who finally has to accept the product. In an agile development process, NFR definition might occur in a later iteration as well.

Regarding our image filtering service, the targeted time performance requirements can be stated clearly by according measures such as response time. Accordingly, interoperability can be defined by stating the interface and protocol standards, that shall be fulfilled by the service.

As a second step, the NFRs have to be prioritized. Just like the first step, this is done in accordance with the customer's demands. In our image filtering example, the customer might assign a very high priority to interoperability, while fault tolerance and correctness are less important. For documentation, only high-priority NFRs are taken into consideration, even if low-priority NFRs are addressed by architecture solutions as well. *Not documenting* the solutions has some obvious drawbacks: these solutions are less likely to be implemented correctly, their implementation can not be validated against a documentation, and new team members can only learn them by reviewing the existing code or by learning from other team members. The benefit of not documenting some less important solutions in favor of others is that developers are more likely to actually use and completely perceive the documentation. Thus, they can understand the architecture's value for the project, accordingly obey the defined architecture constraints and use the documented structures and terms for communication. In this way, the created lightweight documentation disencumbers the architect's job, as less effort is needed not only to maintain the documentation, but also to enforce it.

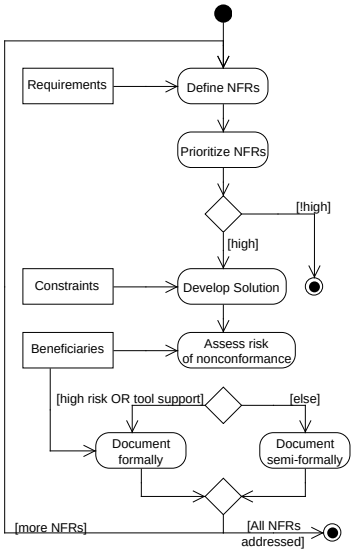

Figure 2: Reducing architecture documentation based on prioritized NFRs

In the third step of the process, the architect develops an appropriate solution for the most important NFRs. This architecture development is highly influenced by the project's constraints such as existing infrastructures, predetermined technologies or project budget. In this way, the architecture decision itself, also the related constraints, are documented; in this way, future architects and beneficiaries will be able to assess and revise these decisions. In our cloud-based image filtering example, the customer might already run a cloud infrastructure which the new service shall be embedded in to save additional costs of operation.

The risk of nonconformance is influenced by two factors, that the architect has to assess, once solutions that fulfill the NFRs of high priority are defined: one factor is the probability that implementations do not conform to the planned solution. As a second factor, he has to assess the different impacts of potential nonconformance. There are many aspects that affect the probability of nonconformance: solutions which are often discussed or need explanation by developers are more likely to cause nonconformance than common sense solutions. Analogously, more complex or nonstandard

solutions bear a higher risk of nonconformance than a simple or standard solution.

The architect has to ensure the correct implementation of solutions that fulfill important NFRs. Accordingly, these solutions should be documented formally, such that available tools for validation can be used. Tool support for creating, editing and utilizing the produced documentation represents an important concern.

In contrast, a *semi-formal documentation* is sufficient for architecture aspects that carry a lower risk of nonconformance. Although the utility of semi-formal documentation for tool based validation is limited, they bring the benefit of low learning efforts. On the other hand, they tend to be ambiguous and incomplete. Furthermore, these typical inadequacies of semi-formal documentation are hard to find, especially for their authors.

Formal documentation demand for a considerable effort to learn its syntax and the handling of associated tools. Additionally, the architect is bound to the expressive power of the chosen language, which possibly does not cover some aspects of the documented solution. Laborious workarounds can be necessary in this situation. However, the according efforts are lowered by the available tool support for creating and editing formal language (see Figure 1). Once a formal solution is set up, it can be utilized to automatically validate an implementation. Due to its' automation, the validation can be conducted earlier in the development process and it can be repeated at the same precision without high efforts. In our exemplary development project for image filtering in a cloud environment, the architect assesses the risk very high, that developers might not fulfill a certain protocol, because they can freely edit the client to communicate with the service in a non-standard way. In this case, a well-documented dummy-client could serve to document the concrete interface in a testable manner. This way, the architect can ensure, that the service can be used by any other client that communicates in conformance with the chosen protocol. For the performance requirements, a short, nonformal documentation might suffice, if the developers are experienced in implementing image processing software and according libraries are already part of their toolbox.

6. CONCLUSIONS AND FUTURE WORK

In this paper we propose a first suggestion about how architecture aspects should be prioritized and if they should be captured in the architecture documentation. The goal is to create software architecture documentation that only captures the most essential architecture aspects w.r.t evolution. For this, we propose a process helping in identifying the most important concerns. We additionally present several demands concerning software architecture documentation. This paper provides a vision towards a more efficient and effective software architecture documentation.

Nevertheless, more work has to be done in order to evaluate the suggested process presented in Section 5. We plan to conduct empirical studies in order to investigate the state of the practice concerning software architecture documentation. In this study, we firstly want to investigate what kind of information is actually captured in a software architecture document, who is using it and which information is actually used from it. Furthermore, the study shall reveal if and how practitioners use a kind of prioritization in order to decide which information should be documented. Additionally, we want to investigate the influence of architecture documentation on a reduction of architecture erosion. In a next step, we want to evaluate our process in an industrial environment. Based on the study results, the prioritization process will be refined.

Beyond the guideline on what is to be documented at what level (see Section 5), we plan to further investigate, which concrete types of semi-formal and formal means of expression and which amount of detail are appropriate for architecture enforcements of different architecture aspects. Thus, we strive to provide clear guidelines for architects to use appropriate architecture documentation instead of using UML-like diagrams or informal boxes and lines at random.

More work has to be invested to documentation activities in agile processes because of the inherent focus on simplification and process efficiency.

7. REFERENCES

[1] S. Ambler. *Agile ModModel. Effective Practices for eXtreme Programming and the Unified Process.* John Wiley & Sons, 2002.

[2] P. Clements, F. Bachman, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures. Views and Beyond*. Addison-Wesley Longman, Amsterdam, 2003.

[3] I. Hadar, S. Sherman, E. Hadar, and J. J. Harrison. Less is more: Architecture documentation for agile development. In *CHASE '13*, May 2013.

[4] A. Jansen and J. Bosch. Software Architecture as a Set of Architectural Design Decisions. In *WICSA '05*, 2005.

[5] I. Malavolta, P. Lago, H. Muccini, P. Pelliccione, and A. Tang. What Industry Needs from Architectural Languages: A Survey. *IEEE Trans. Softw. Eng.*, 2013.

[6] C. Miesbauer and R. Weinreich. Classification of design decisions–an expert survey in practice. In *ECSA 2013*, pages 130–145. Springer, 2013.

[7] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, Oct. 1992.

[8] S. Schröder, M. Riebisch, and M. Soliman. Architecture enforcement concerns and activities - an expert study. In *ECSA '16*, 2016.

[9] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. Ali Babar. A comparative study of architecture knowledge management tools. *J. Syst. Software*, 83:352–370, 2010.

[10] J. Tyree and A. Akerman. Architecture decisions: Demystifying architecture. *IEEE software*, 22(2):19–27, 2005.

[11] R. Wojcik, F. Bachmann, L. Bass, P. Clements, P. Merson, R. Nord, and B. Wood. Attribute-Driven Design (ADD), Version 2.0. Technical report, CMU/SEI, 2006.

[12] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster. Reusable architectural decision models for enterprise application development. In *QoSA '07*, pages 15–32, 2007.