

Maintainability through Architecture Development

Bas Graaf

Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
b.s.graaf@ewi.tudelft.nl

Abstract. This position paper investigates on the need to put software architecture evaluations for maintainability in a broader perspective than is done until now.

1 Introduction

Embedded systems are getting increasingly complex. Consequently developing software for embedded systems becomes more difficult. Nevertheless an industrial survey we conducted confirmed that the embedded systems industry is very cautious in adopting new development technologies [1]. One reason was that some of the industry's practical problems are not solved satisfactory by available technologies.

Industry is rarely developing new products from scratch. This implies that development artefacts can be reused; not only code, but also architectural design. This is one aspect of industrial development that is insufficiently addressed by current approaches. Although more effective reuse is promised by software (product line) architecture technologies, reuse is done ad-hoc in most companies, e.g. by using 'copy-paste'. Therefore these technologies did not yet fulfilled their full potential.

Continuously changing requirements, caused by changes to stakeholder objectives or the environment in which the software is embedded, is another aspect that is insufficiently supported by current technologies. Due to the increased complexity of embedded systems this is very difficult to handle using current development technologies.

Both examples put embedded-specific type of demands on the development of embedded systems. One is related to the capability of a software product to be reused, for which it possibly needs to be (slightly) modified. The other is about the capability of a software product to be adapted to changing stakeholder objectives or environment. Both are closely related and are about the ease of reusing existing software products for new products or for the same (adapted) product. In this paper we use the term maintainability to refer to this desired property of embedded software products.

In this paper we claim that there are already many technologies available to help software engineers to better understand the software architecture and its implications from this maintainability perspective. However, these technologies only shed light on one side of the problem as will be explained in this paper.

2 Related Work

According to [2] different concerns regarding a software system should be addressed in different views on the architecture. A view conforms to a corresponding viewpoint, of which many have been defined [3],[4],[5]. Some of these viewpoints partly address the maintainability concern as explained in the introduction, e.g. the module dependency view.

Architecture description languages provide a formal semantics that makes it possible to analyze the architecture with respect to several properties such as time behaviour. Other design level properties such as liveness, safety, and deadlock can be verified as well. However, it is not easy to see how these properties are related to specific stakeholder requirements. Furthermore these approaches in general only allow for analysis of operational properties of software systems and their applicability to realistic, industrial situations is limited.

Another type of techniques for architecture analysis uses scenarios. Mostly these techniques are based on ATAM or SAAM [6]. By identifying the key-concerns and analyzing the architecture for its support for different scenarios a software architecture can be evaluated. An advantage of these techniques compared to ad-hoc evaluations is that the evaluators do not have to be domain experts. Furthermore the use of scenarios makes it possible to evaluate non-operational attributes, e.g. by using change scenarios. However, the analysis is mainly based on the experience of the evaluators, who typically are highly skilled and experienced architects. This makes people that can do this type of analysis scarce, and as systems become increasingly complex, even more so in the future.

3 Maintainability and Architecture

Some of our observations during the survey we conducted, suggested that it is not always clear what kind of architecture is under consideration: is it the architecture as documented, or as the architects understand it, or maybe as it is implemented. We believe that it is very important to be aware of what is actually considered when evaluating a software architecture for maintainability. This led to the idea that the impact of software architecture on maintainability is easier understood if one considers how each architecture development activity contributes to it. For analysis the same is true: what has been done in each activity to realize maintainability? This idea led to the hypothesis:

The maintainability of a software system can be more effectively evaluated by separately considering three different aspects of software architecture: design decisions, documentation, and implementation.

In Table 1 the three aspects in the hypothesis are clarified by a corresponding question. These aspects are derived from our view on the software architecture development process (Figure 1) and are discussed in the subsections below.

Table 1. Maintainability questions.

Aspect	Question
Design decisions	Does a design decision support maintainability?
Documentation	Can the rationales for a design decision be recovered, e.g. by traceability to requirements?
Implementation	Is this design decision respected in the implementation?

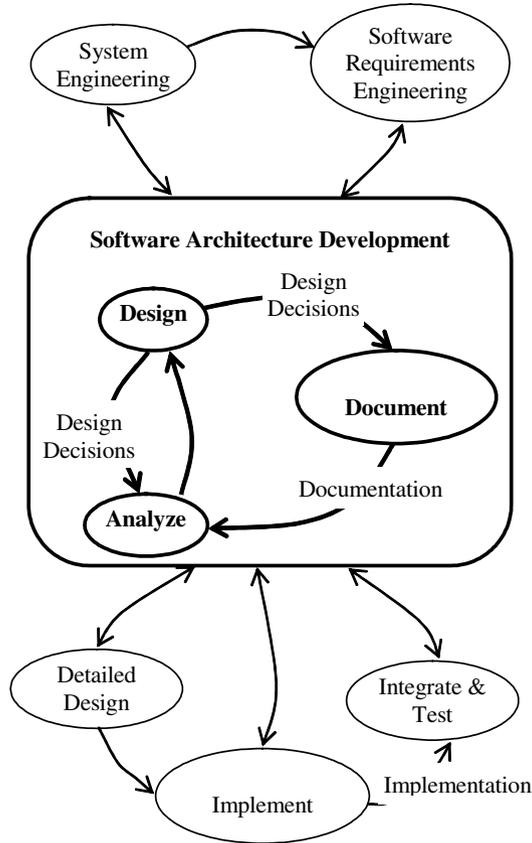


Fig. 1. Architecture development activities.

3.1 Design Decisions

Design is one of the activities in the software architecture development process (Figure 1). During this activity design decisions are taken. Software architecture design is concerned with global design decisions. Little support for this activity is available, besides guidelines in the form of architectural styles [7],[8], patterns [9] or tactics [10]. Consequently its outcome is very dependent on the architect’s skills and experi-

ence. The result of this activity is a virtual architecture of the system. It is virtual in the sense that the decisions are neither documented nor implemented during this activity.

The architectural design decisions taken at the beginning of a project have a significant impact on the maintainability of a software system. For example the decision to merge two components can make maintaining the system more difficult.

In order to raise the confidence that these important design decisions are indeed appropriate, different technologies are available to analyze them, another core activity of architecture development. In practice design decisions are continuously analyzed, most times implicitly. The result of both architecture design and documentation can be used as input for analysis. The extent to which either one is used depends on the type of technique. For instance when architects evaluate for themselves if some design decision is appropriate they will use mostly the outcome of the design activity alone, i.e. the virtual architecture. On the other hand, when applying formal verification techniques, the documentation in the form of models and properties is the main input. Scenario-based techniques use both inputs as they rely on the documentation as well as on the presence of architects and other stakeholders during an evaluation session.

3.2 Documentation

One of the most challenging problems in software architecture is how to achieve a shared understanding of the virtual architecture designed by a small group of architects. Such a shared understanding is important to maintain the conceptual integrity [11] of a design. Therefore, at some point the result of the decisions taken during the architecture design activity have to be documented. This is also shown in Figure 1. The resulting documentation typically consists of diagrams and explaining text. Several architecture description languages and supporting tools are available for the architecture documentation. UML is also often used for this. The documentation serves various purposes, such as communicating design decisions to programmers, or as a basis for analysis of design decisions (Figure 1).

Considering the (documented) design decisions themselves is not enough to make statements about the maintainability of a software system. After all, suppose it is impossible to determine why a design decisions was taken (rationale). That will make it very difficult to assess the impact of changing a design decision. This suggests that the documentation by itself is also very important when considering maintainability.

3.3 Implementation

Besides the virtual architecture as designed by the architects and (partially) documented, there is the implemented architecture. As the designed and documented architecture in principal represent the ‘software-to-be’, the implemented architecture represents the ‘software-as-is’. Due to architectural erosion [12] and miscommunications these are not necessarily the same.

Conformance is the extent to which the implemented architecture corresponds to the virtual architecture and is an important aspect of maintainability. This is illustrated by a project that was considered during the survey we conducted [1]. In this project

the maintainability of architectural components was analyzed by determining the complexity of the components. In fact this is an analysis of the design decisions. By the use of metrics one component was identified as very complex. However, when asked, the architects pointed out a different component to be difficult to change. This component only had a modest score on the complexity metric. Further analysis revealed an unexpected relation in the implementation to another component. This difference between the designed and implemented architecture (i.e. conformance) was an important reason for this component to be difficult to change.

Several techniques are available to ensure this conformance, such as change management, code generation and the use of product family architectures. Furthermore by the use of reverse engineering techniques views on the implemented architecture can be generated, which can be compared to the documented and virtual architectures.

4 Contributions

The contributions of our research will involve software architecture development techniques that address the three aspects discussed in this paper. Furthermore the integration, consolidation, and interpretation of the results of these techniques will be addressed. Hence the result of this research provides more insight in how different aspects of architecture development affect software maintainability. Consequently, it will result in the definition of architectural views that specifically address the maintainability concern. Finally, it will increase the insight in the applicability and tailorability of technologies for architecture development that have been defined.

5 Methods

Besides literature research the ideas in this paper are currently based on discussions we had with over 35 software practitioners during a survey. It was conducted in the context of an ITEA (www.itea.org) project called MOOSE (www.mooseproject.org). This industry driven project aims at improving software quality and development productivity for embedded systems. It offers possibilities to validate ideas and developed or tailored technologies. We pursue a series of small-scale industrial experiments, each addressing a different aspect of architecture development (Table 1). Currently two industrial experiments have been defined. In one of them the behaviour of an architectural component is redocumented. In this case the approach is bottom-up: we try to identify what information engineers need to (re)use or change a component. In the other experiment we consider the effect of design decisions on the maintainability of a reference architecture. By conducting a SAAM-like assessment the reference architecture is evaluated with respect to its support for future product generations. We intend to augment the SAAM with architectural strategies that embody explicit architectural knowledge, such as tactics [10].

Additionally we have defined a small in-house project at our department in which we consider conformance. This project addresses questions such as: how can be ensured that architecture decisions are implemented and how can the conformance be analyzed, i.e. what criteria are relevant when considering conformance from a maintainability perspective.

References

1. B. Graaf, M. Lormans, and H. Toeteneel. Embedded Software Engineering: state of the practice. IEEE Software, November 2003.
2. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. Std 1471-2000, IEEE, 2000.
3. P. B. Kruchten. The 4+1 view model of architecture. IEEE Software, November 1995.
4. C. Hofmeister, R. Nord, and D. Soni. Applied Software Architecture. Addison-Wesley, 1999.
5. P. Clements, F. Bachmann, L. Bass, et al. Documenting Software Architectures. Addison-Wesley, 2002.
6. P. Clements, R. Kazman, and M. Klein. Evaluating Software Architectures. Methods and Case Studies. Addison Wesley, 2001.
7. M. Shaw and D. Garlan. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.
8. F. Buschmann, R. Meunier, H. Rohnert, et al. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley, 1996.
9. J. Bosch. Design & Use of Software Architectures: Adopting and evolving a product-line approach. Addison-Wesley, 2000.
10. F. Bachmann, L. Bass, and M. Klein. Illuminating the Fundamental Contributors to Software Architecture Quality. CMU/SEI-2002-TR-025, Software Engineering Institute, Carnegie Mellon University, August 2002.
11. F. Brooks, Jr. The Mythical Man-Month. Addison-Wesley, anniversary edition, 1995.
12. D. Perry and A. Wolf. Foundations for the Study of Software Architecture. ACM SIGSOFT Software Engineering Notes. October 1992.