Task-Specific Architecture Documentation for Developers

Why Separation of Concerns in Architecture Documentation is Counterproductive for Developers

Dominik Rost(✉) and Matthias Naab

Fraunhofer IESE, Kaiserslautern, Germany
{dominik.rost,matthias.naab}@iese.fraunhofer.de

Abstract. It is widely agreed that architecture documentation, independent of its form, is necessary to prescribe architectural concepts for development and to conserve architectural information over time. However, very often architecture documentation is perceived as inadequate, too long, too abstract, too detailed, or simply outdated. While developers have tasks to develop certain features or parts of a system, they are confronted with architecture documents that globally describe the architecture and use concepts like separation of concerns. Then, the developers have the hard task to find all information of the separated concerns and to synthesize the excerpt relevant for their concrete task. Ideally, they would get an architecture document, which is exactly tailored to their need of architectural information for their task at hand. Such documentation can however not be created by architects in reasonable time. In this paper, we propose an approach of modeling architecture and automatically synthesizing a tailored architecture documentation for each developer and each development task. Therefore architectural concepts are selected from the model based on the task and an interleaving of concepts is done. This makes for example all interfaces explicit, which a component has to implement in order to comply with security, availability, etc. concepts. The required modeling and automation is realized in the tool Enterprise Architect. We got already very positive feedback for this idea from practitioners and expect a significant improvement of implementation quality and architecture compliance.

Keywords: Architecture documentation · Architecture knowledge · Architecture realization · Developers · Implementation · Task · Separation of concerns

1 Introduction

It is widely accepted that architecture documentation is necessary to prescribe architecture concepts and preserve architecture knowledge over time. This is particularly true in complex project settings: When systems are large and have long lifecycles, architecture documentation serves as a tool to preserve the most important design decisions, and to facilitate communication between stakeholders. Also, software

B. Tekinerdogan et al. (Eds.): ECSA 2016, LNCS 9839, pp. 102–110, 2016.
DOI: 10.1007/978-3-319-48992-6_7

development becomes a more and more distributed and globalized activity, often delaying or even making direct communication impossible. In such settings, architecture documentation is a vital communication vehicle to allow a consistent realization of the architecture.

Architecture documentation for such systems can become large. In our experience, for large-scale projects several hundreds of pages are realistic. Working with such documentation can be difficult, in particular for developers, who use it as the basis for their implementation activities, for two main reasons:

First, the perspectives of architects and developers on the system diverge. Architects focus on the system as a whole, designing the overall principles of the system for a multitude of stakeholders. They break down the big and complex problem of the complete system into smaller parts, i.e. apply the principles of divide and conquer and separation of concerns, to create concepts that address architecture drivers in a consistent and uniform way. Examples are concepts for exception handling, validation, scaling, etc. For a medium sized project this can easily lead to 20–50 different concepts.

Our central insight is that *while separation of concerns is vital for architects, who deal with a problem too large to handle as a whole, it is actually counter-productive for a developer working on a task with a narrow focus on single entities, because the separated concerns need to be located and synthesized again*. When developers implement single modules, they need to know and consider several architecture concepts and realize them in their specific context. Such concepts are normally not explicitly described for every element that needs to realize it, but once, in a general way and then instantiated throughout the system (e.g. which interfaces to implement in which way for the security concepts, for transaction handling, …). This means, every single developer needs to be aware of or search for relevant concepts for the development task at hand (cf. Fig. 1).

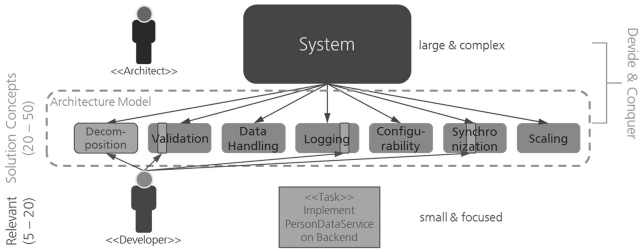

Fig. 1. Architect developer perspective difference

The second aspect is related to the architecture-code-gap [1]. When architects design the system, they reason about the system in terms of components, layers, or decisions. Developers on the other hand work with classes, packages, and interfaces. While it is reasonable for the different roles to work with the elements that best suit their needs, their inherent difference creates an obstacle for architecture realization: There is an additional cognitive step to transform architecture concepts, to the code level.

Both aspects lead to an architecture realization that is on the one hand less efficient, because developers are required to search and identify relevant concepts in a large amount of architecture information. On the other hand, it is error prone because developers under high time pressure might not take the time to consult the architecture documentation, causing architecture violations and consequently architecture erosion [2].

To address these problems, we propose an approach of automatically generating architecture documentation specific for tasks of individual developers. An overview of the approach is presented in Sect. 3 before we describe its details in Sect. 4. To get a better understanding, we present an example in Sect. 5 and conclude in Sect. 6 with validation and future work.

2 Related Work

The approach we present in this paper is built on the foundations of architecture documentation and architecture views. Several different works cover these topics and have presented their own documentation approaches and view sets: [3, 4], etc. Views are a tool for separation of concerns during the design of a software system, but can also be used to tailor information towards the readers [5]. This, however normally refers to types of stakeholders, like developers in general, not more specific.

The idea of considering design decisions as an integral part of architecture and documentation started the whole new research field Architecture Knowledge Management (AKM) Capilla et al. published a comprehensive analysis of the work done in AKM in the past ten years [6]. Our approach is closely related to these approaches, in particular those that provide some kind of personalization mechanism, i.e. making AK specific for a target group. EAGLE [7], ADDM [8], and Decision Architect [9] are examples.. However, none of them tackles the described challenges, either they focus on personalization for stakeholder types, not individuals, or their goal of personalization is different.

The approach we present in this paper is the advancement of the preliminary and basic ideas we outlined in [10]. To align our work with the needs of industry we also performed a comprehensive state of the practice analysis of architecture documentation in industry in [11].

3 Approach Overview

3.1 Task-Specific Architecture Documentation

We frequently experience the challenges we describe in Sect. 1 in industrial projects with our customers. For this reason, we developed an approach for creating architecture documentation that is not only specific for a certain group of stakeholders, but for individual developers and each of their individual development tasks. The resulting architecture documentation centers around the specific architectural elements that developers need to change, create, or delete. It provides detailed information on these *focus elements*, together with all relevant information from architecture concepts that need to be considered. Besides the elements, “relevant information” includes their

internal structure, interfaces to provide, location in the source code, and relations to create. These pieces of information are combined, so that a meaningful view on a very specific part of the system is created. Thus, the architecture documentation for developers contains only a minimum of overhead information, and in a form that allows direct realization. Manually creating such documentation is economically impossible, hence, task-specific architecture documentation needs to be created fully automated with a tool.

3.2 Development Setting and Tooling

Task-specific architecture documentation is not bound to a particular development process, but works best with highly iterative approaches with small increments, like agile development. In each iteration, a user story or use case is selected for realization. For the selected user story, a project manager, architect or even the team derive development tasks. For each of these development tasks, the architect generates the corresponding task-specific architecture documentation, which is used by developers when they carry out the task. Normally, an architect has created an architecture design for all relevant concepts in a previous iteration, so that it is ready for realization in the next.

The architecture model is based on UML and the documentation generator is created as an add-in for the widely used modeling tool Enterprise Architect (cf. Fig. 2). The add-in works on task specifications, that reference elements from the architecture model. Therefore, task specifications are created as elements in the architecture model as well (in future versions an integration with issue tracking systems is planned). The resulting documentation is created as a read-only document. In future versions, the result can be generated as a small, tailored architecture model, that can be integrated with a viewer in the IDE developers normally work with. This allows more interactive working the documentation and more sophisticated linking between documentation and code.

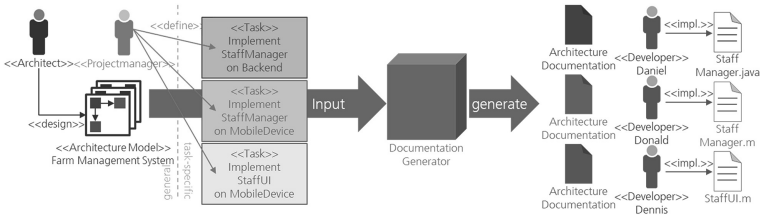

Fig. 2. Development setting and tooling

3.3 Foundational Principles of the Modeling Approach

Our modeling approach has several similarities to many others, it is based on the ISO/IEC/IEEE 42010 standard, uses UML, views, etc. Therefore, transferring the idea to other architecture approaches should be simple. However, two aspects we need to

highlight, that are specific and have an influence on the documentation generation approach. First, when we design architectures, we explicitly *differentiate between runtime and development time*. We often see people drawing boxes and lines, mixing the two arbitrarily, without understanding the differences. Runtime elements (components) can be multiply instantiated and deployed. They are realized with development time entities (modules), which for example represent classes. Different mappings are possible between these entities. Components are normally realized by multiple modules; to optimize reuse, one module can be used for the realization of many components.

The second aspect is *template elements*. They result from the idea of making architecture modeling more efficient by grouping similarities. To eliminate the necessity to describe a concept every time it is applied in the system, template elements are used to describe a concept once. For example, if, as a part of the *validation concept* in a system, we wanted to express that for every *backend service* in the system, there has to be a corresponding *validator* component to validate the data received from clients, we could model the template components *T_Backend Service* and *T_Backend Service Validator* as shown in Fig. 3 and link them to concrete instances. With this idea, the required amount of modeling to describe the architecture concepts of the system can be reduced.

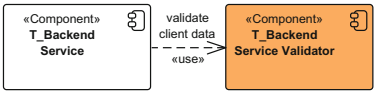

Fig. 3. Service validator concept

Fig. 4. Example of a task specification

4 Detailed Approach

The documentation generation process realized in the documentation generator can be divided into four distinct parts, which are explained in the following sections.

4.1 Task Specification

Development tasks are the starting point for the generation of task-specific architecture documentation. They refer to one or a set of architecture elements to work on and are carried out by a developer. Our approach focusses on the constructive task types, like design and implementation, whereas deeply technical tasks, like bug fixing, which are hardly covered in an architecture model, are out of scope.

Architects model task specifications in the architecture model, so that architecture elements to which the task refers can be directly linked, traced and processed. A task element has a name and a description. It references architecture elements, the *focus elements*, with either a create, change, or delete relationship. And, a task is assigned to a concrete developer who is responsible for carrying it out. Figure 4 shows an example of a task specification in an architecture model.

4.2 Selection

Selection is the automatic process of analyzing the architecture model and identifying model elements that are relevant to consider for the subsequent steps. The starting point for selection is always one or more focus elements. The following elements are included in the selection processes: Developers need to know all details about the *focus elements,* so all occurrences are included with their properties and description. The hierarchy of the element's *template elements* need to be included because all concepts involving the templates are relevant for the focus elements as well. The *mapped development-time elements* are included because they describe how an element needs to be realized. In our modeling approach, *design decisions* are created in the architecture model as well and relevant ones are included in the selection process. Finally, descriptions of all elements and diagrams are included as well.

4.3 Concept Interleaving

Concept Interleaving is the central automatic processing step of integrating all pieces of information, to create the task-specific architecture documentation. It takes the elements from the selection step and extracts relevant information from them to merge it with the focus elements. This includes the following elements, which are first shifted to development time and then interleaved: *Child elements* describe the internal structure of the element to implement and are normally depicted as within an element or have an explicit "part of"-relationship. Child elements of the focus element, its templates and development time elements are considered. *Relation target elements* refer to any element being the target of an outgoing relation. This denotes, which elements to use and create a relationship to. *Interfaces* show the functionality to provide t other elements. Finally, *DT parent elements* denote the location in the source code project.

4.4 Development Time Shift

The collected information should be presented according to the developer's perspective, to facilitate instant realization. The main idea of the Development Time (DT) Shift is to translate elements from an architecture and runtime level to the code and development time level. This is applied for all selected and elements of interleaved concepts.

We differentiate two ways of doing that: DT-Shift with *explicit development time mappings* or *implicit shifting rules*. To ensure uniformity and a clean code structure and to facilitate reuse, architects might decide to prescribe where in the source code the elements of an architecture concept should be created. In this case, an element is mapped with an *explicit* relation to development time elements, e.g. one component to be realized by three classes. The documentation generator simply replaces such elements by the mapped development time elements for the resulting documentation.

In other cases, to save time and reduce complexity, he may also decide to rely on a set of standard shifting rules. Table 1 provides an overview of these.

Table 1. Implicit shifting rules

Runtime element	Development time element
Component (without Subcomponents)	Class
Component (with Subcomponents)	Package
Component (template)	Abstract module
Interface	Interface
Connector	Class
Data	Class

5 Example

The following example illustrates the main ideas of the approach. The context is a farm management system, a system with which farmers manage and plan machines, grain supply, etc. [12]. The next user story to be implemented in the project is managing staff. This includes the tasks to create the according database structure, the user interfaces, etc. One task for a developer is to implement the backend service (cf. Fig. 4).

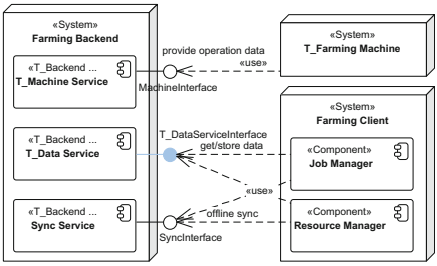

Fig. 5. Example system services

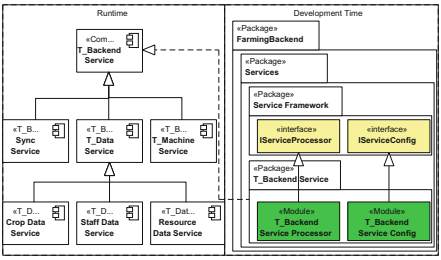

Fig. 6. Example dev. time mappings

Figure 5 shows an overview of the different kinds of services that are provided by the backend. The services that provide the different kinds of data to the applications running on the Farming Client are represented by the template component T_Data Service.

The relevance of this template becomes clear when looking at Fig. 6. The diagram shows the different kinds of services used in the application and an explicit mapping to DT. In this case, for every service, an according package in the services package, together with the processor and configuration classes have to be created. As one example of an architecture concept, Fig. 3 shows a simple validation concept that prescribes every backend service to use an according validator component.

The result of the generation process is depicted in Fig. 7. Colors denote corresponding elements. The focus element Staff Data Service has been shifted to DT according to the explicit mapping shown in Fig. 6, resulting in the Staff Data Service package with the two contained modules. The relation target elements of these two

modules, the two framework interfaces have been integrated. The interface provided by the T_Data Service has been shifted and integrated with a realization relation. The validation concept has been interleaved by adding the shifted validator module. Where possible, the names of templates have been replaced by the name of the focus element.

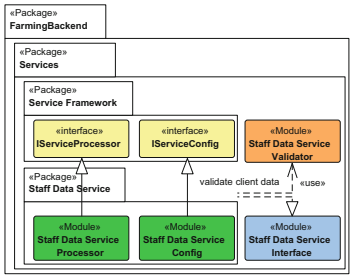

Fig. 7. Example generation result

6 Validation and Future Work

As an applied research organization we work with many industry customers in architecture and development projects. Our experience in these projects and first feedback to the approach gives us good confidence that it will be beneficial to developers in complex development settings. The discussions with our partners show the high demand and positive feedback when we presented our approach. To acquire more formal validation data, we are currently working on a controlled experiment to conduct later this year at Technical University of Kaiserslautern. In this experiment, we will gather data from groups of computer science students, working with task-specific and generic architecture documentation. We will measure the time it takes to identify relevant architecture information for a given development task, as well as errors made when trying to understand the relevant architecture concepts.

In terms of tooling, we are currently developing a prototype according to the conceptual approach presented in this paper and hope to have a running version ready by the end of this year. In the future this is the basis for many possible extensions. For example, IDE integration and feedback mechanisms to architects will provide the opportunity to bring architecture and source code closer together.

References

1. Fairbanks: Just Enough Software Architecture: A Risk-Driven Approach. Marshall & Brainerd (2010)
2. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. ACM SIGSOFT Softw. Eng. Notes **17**, 40–52 (1992)

3. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Addison-Wesley Professional, Boston (2002)
4. Hofmeister, C., Nord, R., Soni, D.: Applied Software Architecture. Addison-Wesley Professional, Boston (1999)
5. Bayer, J., Muthig, D.: A view-based approach for improving software documentation practices. In: 13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems, ECBS 2006, pp. 269–278 (10 p.) (2006)
6. Capilla, R., Jansen, A., Tang, A., Avgeriou, P., Babar, M.A.: 10 years of software architecture knowledge management: practice and future. J. Syst. Softw. **116**, 191–205 (2015)
7. Farenhorst, R., Lago, P., van Vliet, H.: EAGLE: effective tool support for sharing architectural knowledge. Int. J. Coop. Inf. Syst. **16**, 413–437 (2007)
8. Chen, L., Babar, M.A., Liang, H.: Model-centered customizable architectural design decisions management. In: 2010 21st Australian Software Engineering Conference, pp. 23–32. IEEE (2010)
9. Manteuffel, C., Tofan, D., Koziolek, H., Goldschmidt, T., Avgeriou, P.: Industrial implementation of a documentation framework for architectural decisions. In: 2014 IEEE/IFIP Conference on Software Architecture, pp. 225–234. IEEE (2014)
10. Rost, D.: Generation of task-specific architecture documentation for developers. In: Proceedings of the 17th International Doctoral Symposium on Components and Architecture - WCOP 2012, p. 1. ACM Press, New York (2012)
11. Rost, D., Naab, M., Lima, C., Flach Garcia Chavez, C.: Software architecture documentation for developers: a survey. In: Drira, K. (ed.) ECSA 2013. LNCS, vol. 7957, pp. 72–88. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39031-9_7
12. Naab, M., Braun, S., Lenhart, T., Hess, S., Eitel, A., Magin, D., Carbon, R., Kiefer, F.: Why data needs more attention in architecture design - experiences from prototyping a large-scale mobile app ecosystem. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture, pp. 75–84. IEEE (2015)