The Missing Link between Requirements Engineering and Architecture Design for Information Systems

Karl-Heinz Krempels[1,2], Fabian Ohler[1,2] and Christoph Terwelp[1]
[1]*Fraunhofer Institute for Applied Information Technology, Aachen, Germany*
[2]*RWTH Aachen University, Aachen, Germany*

Keywords: Software Architecture, Architecture Design, Design Methodology, Information Systems, Requirements Engineering, Software Engineering.

Abstract: Methodologies for the design of software architectures based on identified system requirements differ in procedure, phases, and artifacts. Furthermore, the consideration of functional and quality requirements during the design process leads to diverse software architecture models varying in their capability regarding adaptivity to new or changed user or quality requirements. The paper discusses a novel methodology for the design of software architecture for information systems based on user and quality requirements. The distinctiveness from methodologies discussed in the literature is given in the comprehensible and traceable deduction of a domain model for the software system architecture from user requirements. The methodology was evaluated and adapted iteratively in many R&D projects.

1 INTRODUCTION

Designing software architectures for large, distributed information systems is a challenging and complex task. To design the software architecture we have to consider a high number of user and quality requirements, the application domain for the developed system, and software architecture design patterns. Therefore, we have to involve domain experts, requirements engineers, system architects, system operators, and users in the design process.

Due to the fact that every existing system is based on an architectural plan and that we have a large number of existing systems, it seems that everybody is able to design a system. However, even if this is the case we can state that differences on architectural structures exist regarding their clear functional description of internal components, design paradigm for interfaces, and the capability of the structure to be used for different distributed systems design paradigms (client-server, service orientation, agent technology, cloud systems, etc.) for implementation and deployment purposes. Due to the complexity of the design task, resulting from the high number of user and quality requirements, the diverse knowledge background of the design team, and the different design methodologies, it seems unrealistic to obtain a comprehensible and traceable software architecture structure with the same user and quality requirements.

So, we can feel that the design process of a good software architecture is also a question of system modeling, design expertise, and the system architects' capability to express metaphorically. The result of a design process is always a software architecture, varying in structural readability, technical applicability, and adaptability to new requirements. Designing software architectures is the art of designing software systems. Our experience in the development of information systems lead to a methodology to design an architecture for large information systems and platforms (consisting of a business model and an information system implementing it). The methodology was designed based on existing findings in the literature and our expertise.

In this paper, we describe the novel methodology for the design of architectures for large software systems. The distinctiveness from methodologies discussed in the literature is located in the comprehensible and traceable deduction of a domain model for the software system architecture from user requirements. To fulfill the given quality requirements the resulting domain model is extended by suitable architectural design patterns for every quality requirement. Section 2 discusses the state of the art for methodologies for requirements and software system engineering. In Section 3, the precursor method is introduced

Krempels, K., Ohler, F. and Terwelp, C.
The Missing Link between Requirements Engineering and Architecture Design for Information Systems.
DOI: 10.5220/0008381701610166
In *Proceedings of the 15th International Conference on Web Information Systems and Technologies (WEBIST 2019)*, pages 161-166
ISBN: 978-989-758-386-5

which was already used and evaluated. Previous evaluation and validation steps are presented in Section 4. In Section 5, the revised methodology is discussed focusing on the procedures leading to the artifacts. Section 6 concludes the paper and addresses future evaluation scenarios.

2 STATE OF THE ART: DESIGNING A SOFTWARE SYSTEM ARCHITECTURE FOR INFORMATION SYSTEMS

Pohl (Pohl, 2010) defines requirements engineering as the process to find out and record what a planned information system should do. He distinguishes among functional requirements, quality requirements, and regulations (technical, legal, organizational, ethical, etc.). ISO25010 (ISO/IEC, 2011) defines the quality requirements categories functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability. Requirement records describe a goal of the planned information system, an interaction scenario between the user and the planned system, an internal scenario among the components of the planned system, or a context scenario without user and information system interaction, but capturing the regulations. These scenarios can be described using natural language, structured lists (enumerations, multi-level enumerations), tables, UML sequence diagrams, UML activity diagrams, or any other suitable modeling language for event and interaction flows.

Studying well known approaches from Agent Technologies (Lind, 2001b; Lind, 2001a; Giorgini et al., 2003; Zambonelli et al., 2003; Weiß et al., 2009) we can state there is a common mapping process between identified requirements for a software system and system services. Lockemann et al. (Lockemann et al., 2006) deduce a design methodology following the divide and conquer paradigm for the design of software architectures for agent systems.

Lichter and Ludewig (Jochen Ludewig and Horst Lichter, 2013) discuss a few architectural design paradigms like information hiding, separation of concerns, and hierarchical grouping. The paradigms are discussed more with the objective to classify and describe resulting software architectures and not as a *how to* instruction to design it.

Sommerville (Sommerville, 2016) discusses requirements engineering, system modeling and architectural design processes. System modeling is discussed following event-driven, data-driven and model-driven paradigms, mixing the software system domain model and the system architecture design process. The out-dated traditional abstraction views for system architectures for information systems, namely architecture in the large and architecture in the small, are also discussed. Sommerville discusses the consideration and implementation of quality requirements in a system architecture using well known architectural patterns for software systems for the requirement of discourse. However, there is a missing link between the requirements engineering phase and the software system engineering phase for an information system, especially a clear methodology.

Finally, in the ISO/IEC/IEEE 12207 Standard (ISO/IEC/IEEE, 2017) describes and defines the software life cycle processes for systems and software engineering. In the architecture definition process, the activities and tasks are defined. However, for the design of the software architecture, the task is defined as *Select, adapt, or develop models of the candidate architectures of the software system.* The methodology is missing.

3 PRECURSOR METHOD

In this section a first approach to address the identified shortcomings of established methods for requirements engineering and software engineering is presented briefly. For a more detailed discussion of the method, see (Beutel et al., 2018b). The additional support for the development of a reference architecture was incorporated as required by the projects the method was to be used in. An overview over the artifacts of this process is given in Figure 1. First actor scenarios (interactions among actors), interaction scenarios (interactions between actors and the system), and system scenarios (system internal interactions) are created in story form (Alexander and Maiden, 2004) to describe the system requirements. Based on the scenarios, user and system tasks are deduced. The tasks are grouped into roles and their relationships. By analyzing possible role distributions on actors, cooperation scenarios are defined. This enables the development of a reference architecture which can be adapted to the identified cooperation scenarios and their hybrids. Based on the cooperation scenarios, the limits of the system and points of contact to external systems can be determined. Taking the system limits into account, the tasks are detailed to activity flows describing the steps required to achieve a system task. The activities are again detailed into function flows focusing on inputs and outputs of functions. The components of the system architecture are

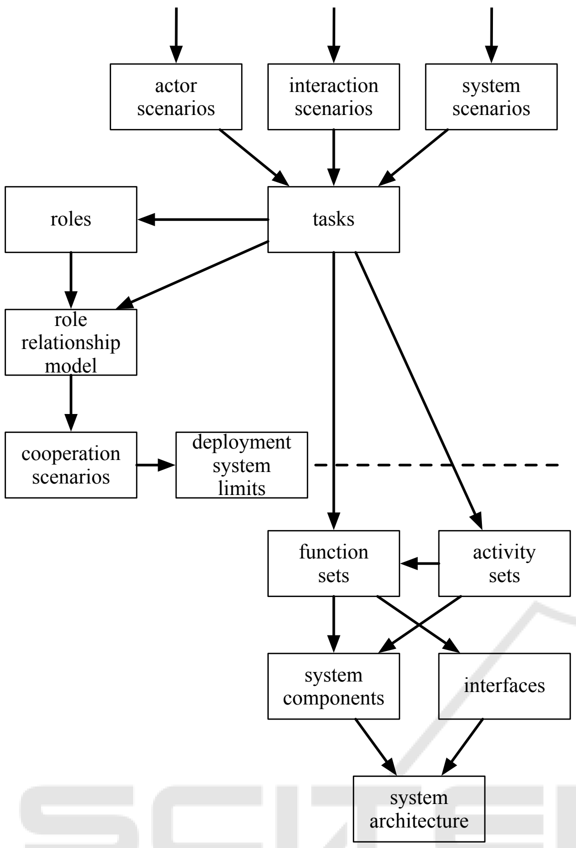

Figure 1: Artifacts of the method used in the development of a reference architecture for open mobility platforms.

derived by grouping the functions by topic and the required information for execution. The interfaces between these components consist of all function calls between them. The combination of components and interfaces form the reference architecture.

4 EVALUATION SCENARIOS

In the projects *econnect Germany* (Krempels, 2015) and *Mobility Broker* (Beutel et al., 2014; Beutel et al., 2016), we realized the importance of separating the functional architecture of an information system from its deployment architecture. Lacking a suitable method in the literature, we developed and applied the method presented in Section 3 during the project *DiMo-OMP* for the development of a reference architecture for mobility platforms (Beutel et al., 2018b). The artifacts of the development process were published in (Steinert et al., 2018) (scenarios and use cases) and (Beutel et al., 2018a) (roles, role relationship model, cooperation scenarios). The resulting components, system architecture, and interfaces are currently in the process of standardization by the association of German transport companies (Verband Deutscher Verkehrsunternehmen VDV) and will be published accordingly. The method and its artifacts were evaluated during several workshops with experts from mobility providers, potential platform providers, large software companies, research institutes, and federal authorities (Terwelp, 2019). The role relationship model, the identified components, their functional description, and the identified interfaces were rated as very important by all participants; they were able to match the concepts to their existing systems and use the artifacts for the planning of future systems. The association of German transport companies recommends the use of the resulting reference architecture in projects developing mobility platforms. Summing up, our approach and results were validated.

Afterwards, we reviewed and refined the method with a special focus on the interaction protocol design phase. The adapted method improves the separation of the functional from the technological aspects of the interaction protocols. It is presented in the next section.

5 PROPOSED METHOD

The method presented in the following aims at developing a system architecture instead of a reference architecture. An integration of the additional steps (e.g., involving cooperation scenarios) might be performed in the future. An overview of the relevant steps and artifacts to be presented is given in Figure 2.

Step 1: Use Case Development. Create use cases (e.g., based on scenarios in story form) and document them in a structured way (e.g., in tabular form or using sequence or activity diagrams). Identify quality requirements (e.g., concerning suitability, performance, efficiency, compatibility, usability, reliability, security, maintainability, and portability) and general requirements.
Main artifacts: use cases, quality requirements, general requirements

Step 2: System Task Identification. Identify the system tasks. Every interaction in the use cases involves a system task. At this point, we have documented *what* the system offers to the user.
Main artifacts: system tasks

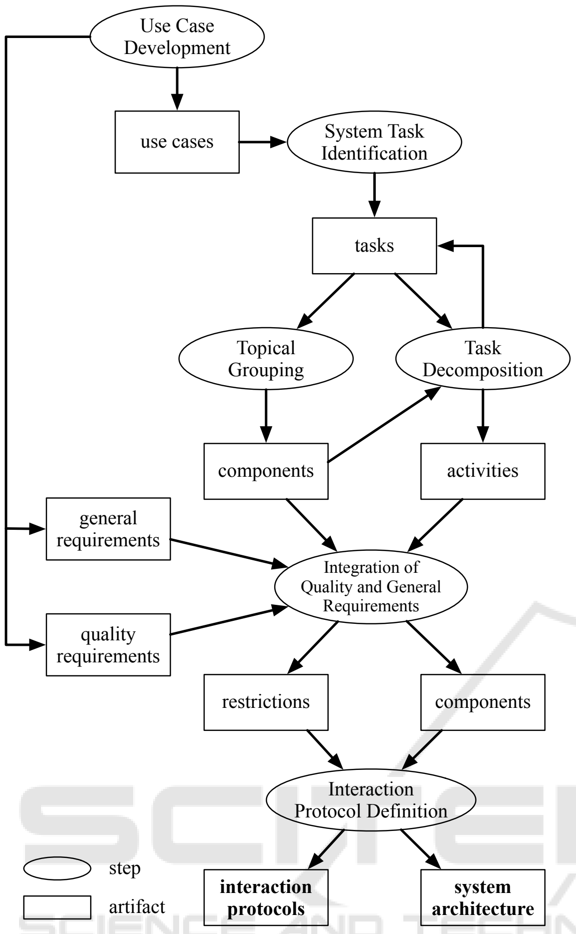

Figure 2: Steps and artifacts of the new method.

Step 3: Topical Grouping. Topically group the tasks to components.
Main artifact: components and their tasks

Step 4: Task Decomposition. This step focuses on the question of *how* the system offers its tasks. Decompose the identified tasks into activities. An activity is characterized by its semantically graspable effect on the system. Analyze the activities with respect to information needs and label them as internal or external to the component. Such a label is based on:

- does the activity belong to the component topically?
- is the information needed available within the respective component when considering all of its tasks?

Define new tasks for the new external activities to meet the information needs or to support the component. In case new tasks are identified, go back to Step 3, otherwise continue with Step 5. The result comprises a functional domain model that should be re-usable for different contexts.
Main artifacts: set of activities and set of external tasks

Step 5: Integration of Quality and General Requirements. Following the 'task dependency graph' resulting from the decomposition of tasks to activities to creating new tasks, propagate the quality requirements (e.g., performance, reliability, security, usability) and annotate tasks appropriately. Tasks used by several other tasks (directly or indirectly) may be annotated with different quality requirements. Thus, the information sources should be noted, too. The quality requirements can, for example, have implications on the viable interaction type: e.g., tasks have to provide a continuous up-to-date stream of information instead of fetching information on-demand.

Well known software architecture design patterns (Sommerville, 2016; Hohpe and Woolf, 2004; Thomas Erl, 2009; Erl, 2008) lend themselves to the integration of quality requirements into the system architecture. Evaluate and apply suitable patterns and solve potentially emerging conflicts weighing the alternatives with respect to the quality requirements.

In doing so, components may need to be annotated with restrictions. These, for example, refer to localization: components have to reside, e.g., on the smartphone of the customer or within the own system borders.

The application of design patterns should be structured in some manner. Aspects influencing the order in which the requirements are addressed are, for example, affected number of tasks, volatility of the requirements in respect to future changes, and coarseness of the applied patterns.

The previous design steps can be done in three different ways:

1. Postponing the general requirements, annotate tasks with quality requirement and apply design patterns to fulfill them.
   Intermediate artifact: domain model satisfying the functional and quality requirements
   Integrate general requirements and apply design patterns to fulfill them.
   Benefit: The intermediate domain model can be used as a basis for different general requirements (e.g., to develop a system used in two different parts of the world).
2. Postponing the quality requirements, annotate tasks with general requirement and apply design patterns to fulfill them.
   Intermediate artifact: domain model satisfying the functional and general requirements

Integrate quality requirements and apply design patterns to fulfill them.
Benefit: The intermediate domain model can be used as a basis for different quality requirements (e.g., in case the quality requirements are expected to be changing more drastically than the general requirements).

3. Integrate general requirements and quality requirements in a single step.
Benefit: Prevents working with the effects of patterns in intermediate domain models that are discarded when considering the postponed requirements.

Main artifacts: quality requirements per task and restrictions per component; components

Step 6: Interaction Protocol Definition. Settle the interactions to respect the requirements regarding the tasks and their encasing components. Define interaction protocols documenting the component interactions. In case different components impose incompatible requirements on a task, different interaction pattern may have to be used alongside each other resulting in multiple implementations.
Main artifacts: interaction protocols and – since the components are known, too – the system architecture

In case the system to be designed is supposed to provide its services to the user via a mobile application, the first two steps consider the interaction of the user with this application and the system tasks are tasks offered by the mobile application. This way, the required flexibility to decide where a component should reside is available in the follow-up steps.

6 CONCLUSION

This paper presents our current efforts to bridge the gap between requirements engineering and software engineering. It presents a refined method to develop a system architecture based on requirements in a comprehensible and traceable way. Our work is supposed to complement the existing literature, which describes the requirements engineering and software engineering activities but leaves out a clear methodology for the design of the system architecture. We thus present the artifacts on the way to the architecture as well as the steps needed to produce these.

Future research involves integrating the steps additional to the development of a *reference* architecture into the method. This has been deferred for evaluation purposes, as most situations demand a system architecture. One is rarely lucky enough to be asked to develop a reference architecture.

We are looking forward to employing and evaluating the proposed method in the EU project *Sharing and Automation for Privacy Preserving Attack Neutralization (SAPPAN)*, in the development of the next version of the electronic fare management system architecture for Germany and in augmenting the results of the project *DiMo-OMP* with data and platform management tasks. In our expectation, the main benefit is the separation of the functional and the technical (deployment, implementation) architecture. This is to be confirmed in the upcoming evaluation. The experience gained during the planned applications of the method will be integrated into a more detailed elaboration of the process steps.

ACKNOWLEDGEMENTS

This work was funded by the German Federal Ministry of Economic Affairs and Energy (BMWi) for the project *Mobility Broker* (Grant Id: 01ME12136), by the German Federal Ministry of Transport and Digital Infrastructure (BMVI) for the projects *Digitalisierte Mobilität – Die Offene Mobilitätsplattform (DiMo-OMP)* (Grant Id: 19E16007B) and *Digitalisierte Mobilität – Fahrzeug und Haltestelle (DiMo-FuH)* (Grant Id: 19E16009G), and by Ford Motor Company for the Projects *Autonomy Mobility Living Lab Aachen* (Grant Id: 160812).

We would like to thank to all the industrial and academic partners from the research projects *Mobility Broker*, *DiMo-OMP*, and *DiMo-FuH* for their cooperation, support, problem related discussions, and finally their great hospitality for project meetings and visionary discussions. Special thanks to Sjef Janssen, Elke Fischer, Michael Bauer, Oliver Waltes, Claus Dohmen, Peter Kehren, Johan van Ieperen, Dirk Weißer, Bert Lange, Tobias Steinert, Werner Kohl, Berthold Radermacher, Detlef Kuck, Markus Beutel, Christian Samsel, David Thulke, Felix Schwinger, as well as all related contributors and project partners.

REFERENCES

Alexander, I. F. and Maiden, N. (2004). *Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*. Wiley Publishing, 1st edition.

Beutel, M. C., Gökay, S., Ohler, F., Kohl, W., Krempels, K.-H., Rose, T., Samsel, C., Schwinger, F., and Terwelp, C. (2018a). Mobility service platforms - crosscompany cooperation for transportation service interoperability. In *Proceedings of the 20th International Conference on Enterprise Information Systems, ICEIS 2018, Funchal, Madeira, Portugal, March 21-24, 2018, Volume 1.*, pages 151–161.

Beutel, M. C., Gökay, S., Jakobs, E., Jarke, M., Kasugai, K., Krempels, K., Ohler, F., Samsel, C., Schwinger, F., Terwelp, C., Thulke, D., Vogelsang, S., and Ziefle, M. (2018b). Information system development for seamless mobility. In Donnellan, B., Klein, C., Helfert, M., and Gusikhin, O., editors, *Smart Cities, Green Technologies and Intelligent Transport Systems - 7th International Conference, SMARTGREENS, and 4th International Conference, VEHITS 2018, Funchal, Madeira, Portugal, March 16-18, 2018, Revised Selected Papers*, volume 992 of *Communications in Computer and Information Science*, pages 141–158. Springer.

Beutel, M. C., Gökay, S., Kluth, W., Krempels, K.-H., Samsel, C., and Terwelp, C. (2014). Product oriented integration of heterogeneous mobility services. In *Intelligent Transportation Systems (ITSC), 2014 IEEE 17th International Conference On*, pages 1529–1534. IEEE.

Beutel, M. C., Gökay, S., Kluth, W., Krempels, K.-H., Samsel, C., Terwelp, C., and Wiederhold, M. (2016). Heterogeneous travel information exchange. In *Internet of Things. IoT Infrastructures: Second International Summit, IoT 360° 2015, Rome, Italy, October 27-29, 2015, Revised Selected Papers, Part II*, pages 181–187. Springer.

Erl, T. (2008). *SOA – Principles of Service Design*. Prentice Hall.

Giorgini, P., Kolp, M., Mylopoulos, J., and Pistore, M. (2003). The tropos methodology: an overview. In Bergenti, F., Gleizes, M.-P., and Zambonelli, F., editors, *Methodologies and Software Engineering For Agent Systems*, pages 1–20, New York. Kluwer Academic Publishing.

Hohpe, G. and Woolf, B. (2004). *Enterprise Integration Patterns*. Addison-Wesley.

ISO/IEC (2011). Systems and software engineering – systems and software quality requirements and evaluation (SQuaRE) – system and software quality models. ISO/IEC 25010, European Committee for Standardization.

ISO/IEC/IEEE (2017). Systems and software engineering – software life cycle processes. ISO/IEC/IEEE 12207, European Committee for Standardization.

Jochen Ludewig and Horst Lichter (2013). *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. Dpunkt.Verlag GmbH.

Krempels, K.-H., editor (2015). *Abschlussbericht der RWTH Aachen zum Verbundvorhaben „econnect Germany”, Stadtwerke machen Deutschland Elektromobil, von Aachen bis Leipzig, vom Allgäu bis Sylt*. Apprimus.

Lind, J. (2001a). The conceptual framework of massive. In Lind, J., editor, *Iterative Software Engineering for Multiagent Systems: The MASSIVE Method*, volume 1994 of *Lecture Notes in Computer Science*, page 97. Springer.

Lind, J. (2001b). Massive views. In Lind, J., editor, *Iterative Software Engineering for Multiagent Systems: The MASSIVE Method*, volume 1994 of *Lecture Notes in Computer Science*, page 121. Springer.

Lockemann, P. C., Nimis, J., Braubach, L., Pokahr, A., and Lamersdorf, W. (2006). Architectural design. In Kirn, S., Herzog, O., Lockemann, P. C., and Spaniol, O., editors, *Multiagent Engineering, Theory and Applications in Enterprises.*, pages 405–429. Springer.

Pohl, K. (2010). *Requirements Engineering - Fundamentals, Principles, and Techniques*. Springer, Heidelberg New York.

Sommerville, I. (2016). *Software Engineering*. Pearson, 9 edition.

Steinert, T., Koreng, R., Mayas, C., Cherednychek, N., Dohmen, C., Hörold, S., Krempels, K.-H., Kehren, P., Kohl, W., Ohler, F., Terwelp, C., van Ieperen, J., and Wiegand, A. (2018). Definition und Dokumentation der Nutzeranforderungen an eine offene Mobilitätsplattform.

Terwelp, C. (2019). *Development of a Reference Model for Mobility Platforms*. PhD thesis.

Thomas Erl (2009). *SOA – Design Patterns*. Prentice Hall.

Weiß, G., Pomberger, G., Beer, W., Buchgeher, G., Dorninger, B., Pichler, J., Prähofer, H., Ramler, R., Stallinger, F., and Weinreich, R. (2009). *Software Engineering – Processes and Tools*, pages 157–235. Springer Berlin Heidelberg, Berlin, Heidelberg.

Zambonelli, F., Jennings, N., and Wooldridge, M. (2003). Developing multiagent systems: the gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3).