Integrating Requirements and Design Decisions in Architecture Representation

Rainer Weinreich[1] and Georg Buchgeher[2]

[1] Johannes Kepler University Linz, Austria
rainer.weinreich@jku.at
[2] Software Competence Center Hagenberg, Austria
georg.buchgeher@scch.at

Abstract. It has been proposed to make architectural design decisions first-class entities in software architecture representation. The actual means of capturing, representing, and managing architectural design decisions is still an open issue of research. We present an approach for capturing requirements and design decisions during design and development. We integrate design decisions, requirements, scenarios, and their relationships along with other architectural elements directly in a single, consistent, and formally defined architecture model. Capturing, visualizing, and tracing of architectural knowledge are supported by an integrated set of tools working on this model. The approach supports comprehensive tracing between requirements, design decisions, other architectural elements, and implementation artifacts, impact analysis, and architecture analysis and evaluation.

Keywords: Software Architecture Models, Design Decisions, Software Architecture Tools, Software Architecture Knowledge Management.

1 Introduction

Design decisions are an important element in software architecture. An early definition of the term software architecture provided by Perry and Wolf [20] already includes rationale in addition to elements and form. According to Perry and Wolf rationale "captures the motivation for the choice of architectural style, the choice of elements, and the form".

Though rationale has already been identified early in the history of software architecture research, it has been neglected in software architecture representation. As Kruchten [16] points out, research in this area has concentrated on representing and documenting a system's architecture from different perspectives, called architectural views. While architectural views and corresponding view frameworks are an important means for documentation, they focus on the result of the design process and lack information about the actual decisions and their rationale [26]. If design decisions are not documented, they remain tacit knowledge [12], which is easily lost [5,26]. Bosch [5] identified the resulting knowledge vaporization as the key problem of design erosion [13] and was one of the

M. Ali Babar and I. Gorton (Eds.): ECSA 2010, LNCS 6285, pp. 86–101, 2010.

first to point out the importance of design decisions in software architecture. He proposed to view software architecture as a composition of design decisions and demanded a first class-representation of design decisions [5].

While capturing design decisions and rationale provides a number of benefits [25], the means for capturing this information is still an open issue of research. Approaches for rationale management exist [11] but suffer from a number of problems. Van der Ven et al. [26] provide some examples like the overhead involved in capturing the required information, and the missing connection between design decisions and architectural elements. In addition, design decisions are closely related to requirements [4,18,21,26]. And requirements, as well as design decisions, are central to many architectural analysis methods [16].

We present an approach for capturing requirements and design decisions and for integrating them in a formally defined architecture representation. A consistent model for requirements, design decisions, and scenarios - called *architectural issues* in our approach - supports architecture knowledge activities like impact analysis and tracing from requirements to architecture elements and implementation artifacts. The same model can be used for architecture analysis and evaluation by connecting architectural issues with analysis state and analysis data. Capturing issues and their relationships is supported as an integrated activity in design and development. The main benefits of our approach are the deep integration of support for architectural issues in design and development, the usage of the same model for architecture knowledge management and architecture analysis, and the integration into a consistent architecture representation, which supports system evolution and enables tracing from requirements to architecture and implementation.

The remainder of this paper is structured as follows: In Section 2 we comment on previous work that has been used as the basis for the work presented in this paper. The section includes an overview of the LISA model, a meta-model for architecture description, and the LISA toolkit. The section also includes references to previous work where appropriate. In Section 3 we give a conceptual overview of our approach. This includes a conceptual model of requirements, design decisions, and scenarios, their relationships to each other, as well as relationships to other architectural elements. In Section 4 we describe tool support for three important aspects of our approach in more detail: capturing, visualizing, and analyzing requirements, design decisions, and scenarios. Section 5 describes the steps we have taken to validate our approach. In Section 6 we present related work. Section 7 summarizes the main aspects of our work.

2 Previous Work

The work presented in this paper is part of the *Software Architecture Engineering* project for supporting architecture-centric software development [7]. The project aims at supporting architecture-related activities like modeling, documenting, and analyzing software architectures in an integrated and incremental way. Integrated means that architecture-related activities are integrated seamlessly in all

software development activities, from analysis to design, and implementation. Incremental means that we aim to provide support for both agile and non-agile project settings with potential interleaved analysis, design, and implementation activities.

The main results of the project are the LISA model, a meta-model for software architecture representation, and the LISA toolkit, which is a set of tools for working on LISA architecture models.

The LISA model has been designed for ease of integration and synchronization with a system's implementation to prevent architectural erosion and architectural drift. It provides not only components and connectors for describing dynamic system structures and configurations, but also lower-level abstractions for describing packages, classes, and modules. These parts of a LISA model can be extracted from and easily synchronized with a system implementation and enable us to support dependency analysis as provided by typical software architecture management tools (AMTs) like Lattix, SonarJ, and Structure101. In this sense LISA combines the concepts of lower-level AMTs, i.e., strong tool support and tight implementation integration, with the concepts of higher-level architecture representation and analysis as supported by architecture description languages (ADLs).

The LISA toolkit is integrated in the Eclipse IDE. Architecture modeling, analysis, and implementation can be performed incrementally and interleaved [27]. The toolkit supports multiple architectural views, which are derived from a single, consistent LISA-based architecture model. Using a single model avoids inconsistencies among views. Architecture/implementation synchronization is supported through continuous forward and reverse engineering [6]. Developers always have an up-to-date architecture description available, which acts as a blueprint for the implementation.

Technology independence is achieved through technology-specific binding models [6]. Currently we support bindings for languages like Java and C# and for component models and technologies like Spring, OSGi, EJB, and SCA.

Both the LISA model and the LISA toolkit are extensible. The LISA model is based on XML-Schema and can be extended with additional sub-models. In this sense it is similar to xADL [9]. The LISA toolkit can be extended with additional architectural views for documentation and visualization, with additional constraints for architecture analysis and validation, and with additional components for architecture/implementation synchronization.

The approach has been developed to support different analysis approaches in one single consistent environment. This includes automatic analysis approaches as offered by ADLs and architecture management tools, and manual analysis techniques like scenario-based evaluation methods.

3 Conceptual Overview

A LISA-based architecture description is organized in modules. Modules are the units of deployment and versioning. LISA-modules can be bound to and deployed

with implementation modules in different implementation technologies. This way an architecture representation can be deployed with an implementation.

LISA modules contain architectural elements and relations between these elements. Examples for architectural elements are classes, components, ports, connections, layers, features, configurations, and systems. Architectural elements can have assigned attributes, which are a kind of high-level specification of semantics that is necessary for validation and verification of certain system properties. Architectural elements can be bound to implementation artifacts through technology-specific implementation bindings. The synchronization engine of the LISA toolkit uses these implementation bindings for keeping architecture description and implementation synchronized. Synchronization is performed incrementally, at each change to either the architecture description or the implementation.

Following the model described above, design decisions, requirements, and their relations are also architectural elements in our model and are part of LISA architecture modules. This means that requirements and design decisions are captured as first-class elements of the architecture representation.

Fig. 1 shows design decisions and requirements in the context of the LISA approach. The LISA toolkit provides different editors and views on a LISA architecture model. The model can be bound to an implementation. As shown in the figure, requirements and design decisions are captured as part of a LISA architecture model. We will describe later how requirements and design decisions are represented in the LISA model, and how requirements and design decisions can be captured and described. For now, we will focus our description on how requirements and design decisions can be related to other architectural elements of a LISA architecture description. As shown in Fig. 1, the starting point for architectural decisions are usually architecturally significant requirements (ASRs). Architecturally significant requirements are requirements upon a software system, which influence its architecture [19]. Like other requirements on a software system, ASRs are typically captured during requirements analysis. Nuseibeh [18] and Pohl/Sikora [21] point out that it may also make sense to capture requirements incrementally and interleaved with architecture design for certain kinds of systems, particularly for innovative systems. We support both scenarios. We support importing requirements from issue management systems for requirements that have been captured beforehand with other tools. For incremental analysis and design we support capturing requirements and design decisions during design and implementation.

As shown in Fig. 1 requirements may lead to design decisions, which may lead to subsequent design decisions. In fact, design decisions may act as requirements for subsequent decisions, which blurs the line between requirements and design decisions. De Boer and van Vliet [4] discuss the similarity between requirements and architectural design decisions and even state: “architecturally significant requirements are architectural design decisions and vice versa”. In our model, requirements and design decisions are both modeled as special kinds of issues. The difference is mainly the kind of description and the source of the issue. Requirements are usually the result of analyzing the problem space, while design

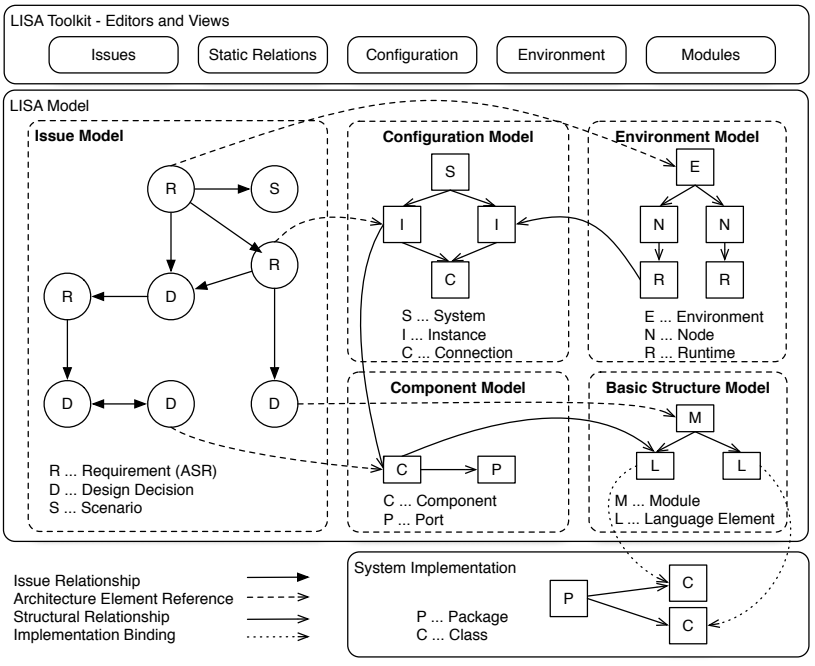

Fig. 1. Requirements and Design Decisions in LISA

decisions are the result of exploring the solution space. As can be seen from Fig. 1, design decisions may also uncover new requirements. For this reason, there may exist a directed relationship from design decisions to requirements in our model.

The defined relationships allow comprehensive tracing. As shown in Fig. 1 requirements and design decisions can be traced to other architectural elements, and through implementation bindings, even to potential implementation artifacts. This depth of integration of requirements and design decisons is also reflected in our toolkit, which supplies markers in an implementation indicating requirements and design decisions that would be affected by changing a particular implementation artifact (see Section 4). Implementation artifacts are not only code fragments but also configuration files [6].

Fig. 2 shows in more detail how requirements and design decisions are modelled in our approach. The central abstraction for representing both requirements and design decisions is an architectural issue. Requirements and design decisions are just special kinds of issues. This reflects the close relationship between requirements and design decisions mentioned before. As can be seen from the figure, issue kinds have not been modelled as subclasses but as attributes of the issue class. This has only technical reasons and facilitates changing an issue kind at run-time without compromising already existing dependencies.

Issues have a summary attribute providing a short description of the issue. The summary is essentially the issues logical name. More detailed descriptions and rationale can be provided as part of specific issue kinds. Issues can be architecturally

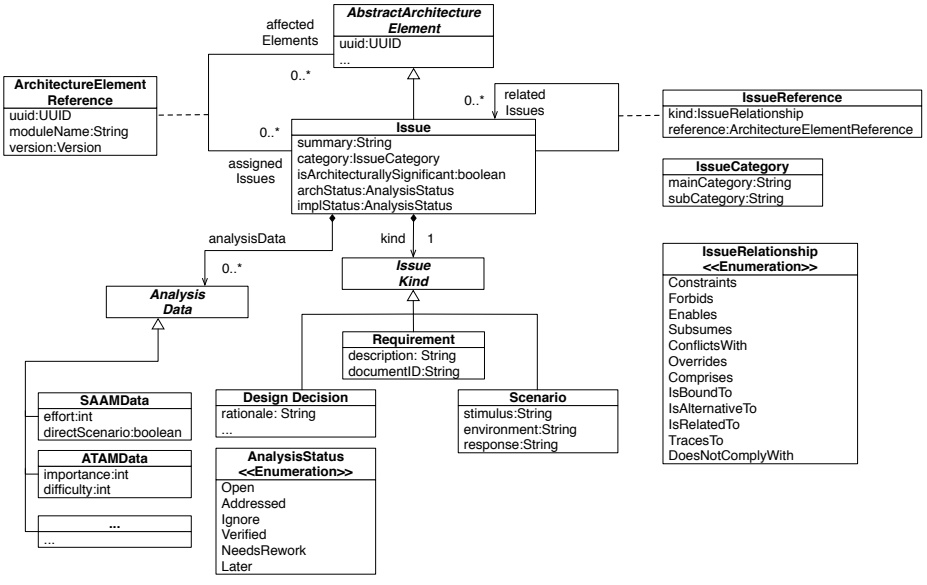

Fig. 2. Requirements Sub-Model

significant. We introduced this field because we also support capturing and describing requirements and design decisions that are not architecturally significant - though the focus of our work lies on ASRs and ADDs. Issues can be assigned to a category, which consists of a main category and a sub category. The category can be used for sorting issues and for automatically building a utility tree as provided by the Architecture Tradeoff Analysis Method (ATAM).

All issues support references to related issues and references to affected architectural elements. References to related requirements and design decisions have been identified as an important element of design decisions [25]. As shown in Fig. 2 we support different kinds of relationships, which are essentially modelled after a taxonomy for design decisions and requirements that has been proposed by Kruchten in [17]. Relations are also used for capturing potential alternative decisions. Capturing alternative design decisions is important for preserving architectural knowledge. Van der Ven et al. [26] even define design decisions as "A description of the choice and considered alternatives that (partially) realizes one or more requirements". Dingsøyr and van Vliet [10] also list alternatives as an important element of a design decision. The second kind of reference that is visible in the model are references to architectural elements that are affected by a change of a requirement or a decision. These references are used for tracing and impact analysis as described above.

A third kind of issue shown in Fig. 2 is *Scenario*. A scenario is a special kind of requirement that enforces a specific description. Kazman [15] defines a scenario as "a brief description of a single interaction of the stakeholder with a system". A scenario is similar to a use case but encompasses the interactions of multiple

stakeholders as opposed to the user only. Scenarios may encompass many requirements [15] and requirements may be derived from scenarios. Scenarios are particularly useful for architecture evaluation [3]. Since scenarios are a special kind of issue, relations from scenarios to other issues and architectural elements can be defined to support tracing and impact analysis.

All three kinds of architectural issues can be used for architecture analysis and evaluation. For this reason, each issue can have associated analysis data and analysis state. Analysis data is used for associating analysis-specific attributes with an architectural issue, like priority and cost. Specific analysis data types can be provided for different architectural analysis methods like SAAM and ATAM as shown in Fig. 2. Each issue has two kinds of analysis state, since evaluating architectural issues is a two-step process. First, it has to be checked that a requirement, design decision, or scenario has been addressed correctly in the architecture. A second step is used for checking the implementation. The initial state of each issue is "open". If an issue has been addressed in architecture or implementation, its status is changed to "addressed". A manual or automatic analysis step ensures that the issue has been addressed correctly. If it has been addressed correctly, the status is changed to "verified".

Currently we support mainly manual analysis. This means that status updates have to be provided by hand. We are currently working on integrating automated analysis on the basis of configurable rules and predefined architectural knowledge. The aim is to provide a combined approach and to enhance and replace manual analysis through automated analysis where possible.

4 Tool Support

Capturing and visualizing architectural issues is supported by the LISA toolkit. The toolkit provides a set of plug-ins that integrates seamlessly with the Eclipse IDE. Editors for defining and visualizing architectural issues are presented in the same environment as other architecture and implementation editors as shown in Fig. 3. The central architecture dashboard shows the modular organization of a LISA-based architecture representation and is depicted in the lower left part of Fig. 3. The main area depicts a form-based editor for capturing information about a particular architectural issue. The lower right part of the figure shows a global or context-specific list of requirements and design decisions, along with issue life-cycle information.

4.1 Capturing Architectural Issues

We provide several options for capturing architectural issues. A central aim has been to support capturing as an integrated activity during design and implementation without the need for switching tools. In the following, we briefly describe the supported means for capturing requirements, design decisions, and their relationships.

Importing Issues from Issue Management Systems. Requirements that have been defined during requirements analysis and specification can be imported

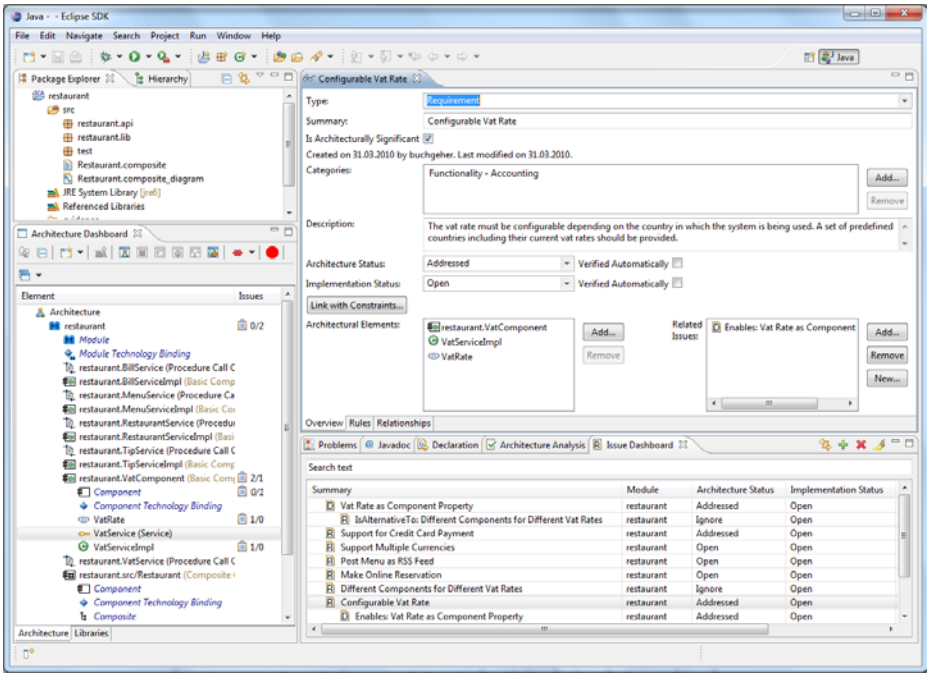

Fig. 3. Issue Views and Editors in the LISA toolkit

from issue management systems. We have implemented an import component for Eclipse *Mylyn*, which acts as a front-end for multiple issue management systems like JIRA and *Bugzilla*. Currently, only importing requirements is supported. A synchronization component supporting two-way synchronization between issue management systems and architecture description is a topic of future work.

Creating Issues Manually. Issues can be created and edited using the form-based editor shown in Fig. 3. The editor is used for manually entering attributes like summary, description/rationale, and category. It can also be used for explicitly defining relationships between issues and between issues and other architectural elements, like components and modules.

Defining Relationships. Relationships can easily be created by dropping related elements onto the corresponding fields in the issue editor. An issue can also be created directly for an architectural element by selecting the element and choosing the "create issue" entry from the provided context menu. This implicitly creates a relation between architectural element and issue and eliminates the need for creating this relationship explicitly. Once an architecture element has been assigned to an issue - be it a requirement, design decision, or scenario - the toolkit also automatically proposes relations to other architecture elements that might be affected by this issue. The proposed elements are determined by analyzing existing relationships between architecture elements.

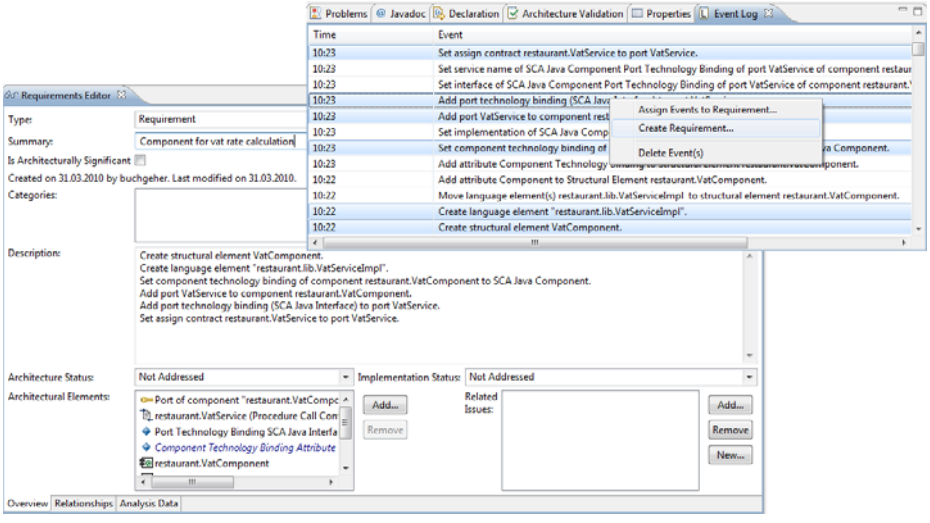

Fig. 4. Creating Issues from Design Activity Logs

Creating Issues from Design Activities. During architecture design, all modifications of the architecture representation are logged. Log entries contain a description of the modification and a list of the architecture elements that are affected by the modification. New requirements and design decisions can be created from the logged activities as shown in Fig. 4. Information about the logged activities is added as part of the description of the new issue and architectural elements that are part of the performed modifications are automatically associated with the new issue. Information from logged activities can also be added to existing issues.

4.2 Visualizing Architectural Issues

Kruchten [16] shows some options for visualizing a set of design decisions, including tables and graphs. Tables lack information about relationships among decisions and graphs may easily become very complex, even for a small set of decisions. For this reason, mechanisms for dealing with this complexity like eliding, filtering, focusing, and sequencing are necessary [16]. In our case, not only design decisions and their relationships are visualized but also requirements, scenarios, and relations to other architecture elements. However, the main challenges of reducing the inherent complexity remain the same. In the following, we provide a short overview of the means for visualizing and editing architectural issues and their relationships in our approach.

Issue Dashboard. The issue dashboard (see Fig. 3) shows all issues that have been defined in the architecture description. Issues can be sorted by name, type, containing module or category. It is also possible to search for specific issues (filtering). A global issue list shows for each issue the relationships to other

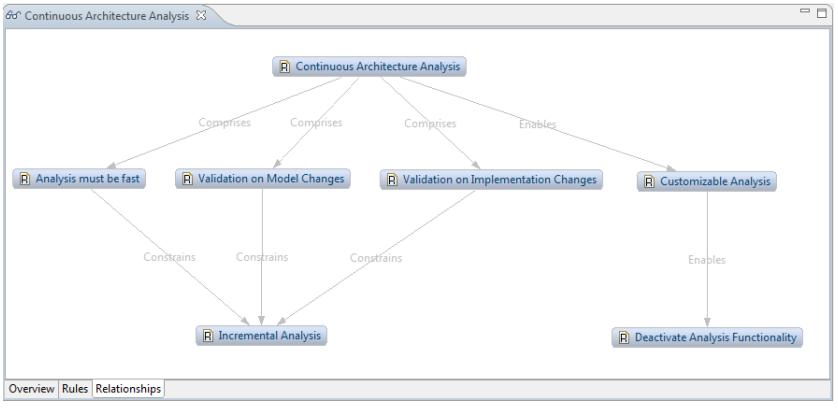

Fig. 5. Dependency Graph

issues. The dashboard can also be configured to show only issues for currently selected architecture elements (focusing). When a user selects an architecture element in one of the architecture diagrams, the dashboard shows all issues that have been assigned to this element. New issues can also be created for selected elements and are automatically assigned to the selected element. The dashboard can also be used for changing the analysis status of an issue without opening the editor for this issue.

Dependency Graph. The issue editor contains an extra page for visualizing the relationships between issues as a directed graph as shown in Fig. 5. This way it is possible to explore the relationships from the viewpoint of a particular issue (focusing) instead of viewing a global relationship graph.

Issues as Part of Architecture Diagrams. Architecture elements that have assigned issues are annotated with a note marker that informs the user that issues have been assigned to that element (see Architecture Dashboard in Fig. 3). The marker also shows the number of assigned issues. Information about the assigned issues is shown as a tool tip.

Resource Markers. Issues can be linked to architecture elements. In turn, architecture elements can be bound to implementation artifacts like source and configuration files through implementation bindings. This enables tracing issues to implementation artifacts. Issues that are related to an implementation artifact are shown as markers in implementation editors. For example, markers are shown in source code editors as icons with tool tips and inform developers about requirements and design decisions that apply to the currently edited implementation.

4.3 Analyzing Architectural Issues

Architectural issues are also the basis for certain kinds of architectural analysis. We mentioned in Section 2 that we integrate multiple approaches for architecture analysis in one consistent environment. Analysis of architectural issues is typically

used for two kinds of analysis: for continuously analyzing and validating that issues are addressed correctly in architecture and implementation and for scenario-based architecture evaluation using architecture evaluation methods like ATAM and SAAM. In general, we aim at supporting incremental and continuous analysis. While dependency analysis and analysis of architecture/implementation conformance are performed automatically, analysis of architectural issues is currently not automated and has to be performed manually.

As described in Section 3, analysis and validation of architectural issues is supported through life-cycle attributes and through analysis data that can be attached to architectural issues.

Life-cycle attributes indicate the analysis status of an issue in the architecture on the one hand and in the implementation on the other hand. The status of analysis in both architecture and implementation is shown in the issue editor and in the issue dashboard. Issues that have not been addressed and issues that have not been addressed correctly are visualized as architecture problems in the architecture diagrams provided by the LISA toolkit.

Analysis data is used for supporting particular architecture evaluation methods. Analysis data can be captured in separate panes for each issue. Additional views can be added for method-specific visualizations. For example, for ATAM we support capturing importance and difficulty, which are used for prioritizing scenarios. The architecture utility tree can be generated automatically based on the defined scenarios and their categories. We also support partly automated analysis like the detection of possible sensitivity and tradeoff points by analyzing architecture elements that have multiple scenarios assigned.

We are currently working on support for automatic analysis of architectural issues on the basis of user-defined element attributes and constraints. The main idea is to support a user in defining issue-specific validation criteria using role-attributes that are assigned to architectural elements.

5 Validation

The benefit of connecting requirements, design decisions, solution structures, and implementation, has been argued widely and is also shown in several approaches and case studies (see section on related work). Main questions that remain are how the required information is captured efficiently and how the captured information can be made easily accessible. We view the main contribution of our approach in reducing the overhead of capturing architectural knowledge and the required relations through integration in one consistent model and environment and in ensuring the consistency of architectural knowledge, representation, and implementation. Therefore, the main aim of validation has been to check the usefulness and usability of the presented approach in practice.

In order to validate the approach we used LISA in a small industrial project for the development of a medical information system. The project was scheduled for one and a half person years, included three developers, and followed a SCRUM/XP process. At the beginning of the project we briefed the users (developers, project leader/architect) in using the toolkit. We gave an overview of

the provided functionality and its intended use. We focused on the supported means for capturing and analyzing requirements and design decisions. We also supported the users during the development process by answering their questions. In order to collect relevant information we observed the usage of LISA with a dedicated monitoring plug-in that logged user interactions in terms of issues captured, including capturing time and owner. Additionally, we conducted interviews with the users to find out about their experiences and the perceived usefulness of our approach. Finally, we reviewed the created architecture description to determine the kind of information captured by users.

Findings. Initially developers had reservations regarding the usefulness of our approach. A particular concern was the additional effort involved in capturing requirements and design decisions. This applied especially for requirements that had already been specified in a separate requirements document. After using the tool for some time this attitude changed and parts of our approach were considered useful. The central benefit perceived was the linking of issues to architecture elements. Capturing requirements and design decisions gave additional meaning to architecture elements. It turned out that users often captured requirements after they had created corresponding architecture elements.

The users doubted the usefulness of defining relationships between issues. Even more, they did not see any benefit from specifying these relationships. They also did not understand the different relationship kinds. One user complained that there "are too many of them and one does not know which to choose". Analyzing the captured requirements and their relationships revealed that primarily functional requirements had been captured. Relationships were typically used for splitting coarse-grained requirements into multiple smaller requirements. In addition, users were unsure about the difference between requirements and design decisions.

What was good? The users particularly liked the seamless integration into the IDE, which facilitated the capturing of requirements and design decisions (as well as the creation and maintenance of the entire architecture description). The issue editor was perceived similar to existing issue management systems, which raised its acceptance among the project team. The general usability of the toolkit was perceived as good with some space for future improvements.

What needs to be improved? The users asked us to provide additional diagrams for visualizing and analyzing the relationships between issues and architecture elements. Particularly they asked for a diagram showing which architecture elements are affected by a set of issues. Currently we only support tracing from one issue to related elements. The users missed additional fields for defining references to other artifacts like existing requirement documents, project guidelines and issues/bugs-ids.

Observations of the research team. IDE integration and traceability from requirements to architectural artifacts have been viewed as the main benefit by the users in the conducted case study. Despite our efforts in reducing the overhead in capturing information, users neglected the knowledge management features. We attribute this mainly to the lack of immediately perceived value in capturing

this information. We think that additional research in making these benefits more clearly might raise the acceptance for such an approach. The support for architecture analysis has not been examined in the described case study.

6 Related Work

Our work combines aspects of architecture description languages (ADLs), software architecture management tools (AMTs), and of tools for software architecture knowledge management (AKM).

ADLs and AMTs focus on automatic architecture analysis. Typical ADL-based approaches lack the level of implementation and IDE integration that is provided in our approach [6]. AMTs provide this integration, but they lack higher-level architectural abstractions and support for analyzing particular quality attributes as supported by ADLs. Representatives of both approaches usually provide no support for capturing and managing requirements and design decisions. The main topics presented in this paper, i.e., capturing and managing requirements and design decisions, support for tracing and impact analysis, and support for scenario-based architecture analysis and architecture evaluation are usually offered by architecture knowledge management tools. A variety of AKM tools exist. Most of them are research prototypes.

PAKME [1] is a web based architecture knowledge management tool, which supports capturing of design decisions and scenarios. It also supports scenario-based architecture analysis. Contrary to our approach, requirements and design decisions are managed independently from architecture representation. Also, PAKME provides no integration with other kinds of analysis and is not integrated in an IDE. The lack of integration leads to overhead for capturing and keeping captured information synchronized with architecture description and implementation. The additional workload through duplication of requirements has been identified as potential problem in using PAKME [2].

Archium [14] also integrates design decisions with architecture description and implementation. Aside from capturing design decisions as first-class entities to prevent knowledge vaporization, the approach also aims at keeping architecture and architectural knowledge consistent during system evolution. This is achieved by integrating a design decision model, an ADL-like architecture model, a composition model, and an implementation in one language (an extension to Java). Archium provides a textual representation, a compiler and a runtime system. This is already an important difference to our approach, which binds and synchronizes an architecture model with an implementation but is independent from particular implementation technologies and run-time systems. From a conceptual viewpoint Archium takes current definitions of software architecture literally; it requires designing software systems by composing design decisions and thus requires a new way of thinking during design. We integrate design decisions differently. Rather than composing a design from design decisions, we bind design decisions to design solutions. Support for architectural analysis and IDE integration as supported in our approach is not available in Archium.

SEURAT [8] is an approach for rationale management in software development, which also provides integration in the Eclipse IDE. Rationale is captured in a semi-formal representation (RATSpeak), which supports the description of decisions, alternatives, and relations to requirements and implementation. The approach supports traceability from requirements and decisions to code. Contrary to our approach, SEURAT provides no architecture representation but connects rationale directly with source code. The authors mention integration with UML as a topic of future work to support rationale management in other software development phases.

AREL [22,24] is an extension to UML. It adds stereotypes for representing architecture elements (AE) and architecture rationale (AR) to UML diagrams. This way rationale and relations to architectural elements are directly modeled in UML using a standard UML tool. Forward and backward traceability is supported through an additional tool, which is implemented in .NET. AREL lacks the deep integration of support for architectural knowledge management into development and design tools that is supported by our approach. This is mainly due to the use of a standard UML tool ([23] p. 201). Also, the combination of architectural knowledge with architectural analysis and validation is not provided.

7 Conclusion

Our approach integrates requirements and architecture design decisions with single consistent architecture representation, the LISA model. A LISA architecture model is continuously checked for consistency and supports continuous synchronization with a potential system implementation. Therefore, the approach supports consistency of the captured requirements and decisions with architectural structures and implementation. Support for consistency is important in wake of incremental development and system evolution. The approach also supports forward and backward tracing from requirements to architecture and implementation and vice versa. We aimed at reducing the overhead in capturing architectural knowledge through IDE integration, through support for defining issue relations by capturing information from design activities, and by suggesting additional relations on the basis of existing ones. Visualization of architectural issues is supported through specific views with support for focusing and filtering. Architectural issues are also presented in other views on architecture and implementation creating an ongoing awareness of related requirements and design decisions during design and development. We have also integrated support for analysis and validation of captured architectural issues and for scenario-based architecture evaluation in our model and toolset. Finally, we have conducted a case study for validating the approach in an industrial project. Integration of architectural issues with architecture representation and implementation and the support for tracing have been well received. Other knowledge management features, like capturing more complex relations between architectural issues, failed to show immediate value to users of the approach. Making the benefits of particular relations more explicit is a necessity for raising the acceptance of such an approach.

References

1. Babar, M.A., Gorton, I.: A tool for managing software architecture knowledge. In: SHARK-ADI 2007: Proceedings of the Second Workshop on SHAring and Reusing Architectural Knowledge Architecture, Rationale, and Design Intent, p. 11+. IEEE Computer Society, Washington (2007)
2. Babar, M.A., Northway, A., Gorton, I., Heuer, P., Nguyen, T.: Introducing tool support for managing architectural knowledge: An experience report. In: IEEE International Conference on the Engineering of Computer-Based Systems, vol. 1, pp. 105–113. IEEE, Los Alamitos (2008)
3. Babar, M.A., Zhu, L., Jeffery, R.: A framework for classifying and comparing software architecture evaluation methods. In: ASWEC 2004: Proceedings of the 2004 Australian Software Engineering Conference, p. 309+. IEEE Computer Society, Washington (2004)
4. de Boer, R., van Vliet, H.: On the similarity between requirements and architecture. Journal of Systems and Software (November 2008)
5. Bosch, J.: Software architecture: The next step. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol. 3047, pp. 194–199. Springer, Heidelberg (2004)
6. Buchgeher, G., Weinreich, R.: Connecting architecture and implementation. In: Meersman, R., Herrero, P., Dillon, T. (eds.) OTM 2009 Workshops. LNCS, vol. 5872, pp. 316–326. Springer, Heidelberg (2009)
7. Buchgeher, G., Weinreich, R.: Software Architecture Engineering. In: Buchberger, et al. (eds.) Hagenberg Research, pp. 200–214. Springer, Heidelberg (2009)
8. Burge, J.E., Brown, D.C.: Seurat: integrated rationale management. In: ICSE 2008: Proceedings of the 30th international conference on Software engineering, pp. 835–838. ACM, New York (2008)
9. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: A comprehensive approach for the development of modular software architecture description languages. ACM Trans. Softw. Eng. Methodol. 14(2), 199–245 (2005)
10. Dingsøyr, T., Vliet, H.: Introduction to software architecture and knowledge management. In: Ali Babar, M., Dingsøyr, T., Lago, P., Vliet, H. (eds.) Software Architecture Knowledge Management, ch. 1, pp. 1–17. Springer, Heidelberg (2009)
11. Dutoit, A.H., McCall, R., Mistrík, I., Paech, B. (eds.): Rationale Management in Software Engineering: Concepts and Techniques. Springer, Heidelberg (2006)
12. Farenhorst, R., Boer, R.C.: Knowledge management in software architecture: State of the art. In: Ali Babar, M., Dingsøyr, T., Lago, P., Vliet, H. (eds.) Software Architecture Knowledge Management, ch. 2, pp. 21–38. Springer, Heidelberg (2009)
13. van Gurp, J., Bosch, J.: Design erosion: problems and causes. Journal of Systems and Software 61(2), 105–119 (2002)
14. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: 5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005, pp. 109–120. IEEE Computer Society, Washington (2005)
15. Kazman, R., Carrière, S., Woods, S.: Toward a discipline of scenario based architectural engineering. Annals of Software Engineering 9(1), 5–33 (2000)
16. Kruchten, P.: Documentation of software architecture from a knowledge management perspective design representation. In: Ali Babar, M., Dingsøyr, T., Lago, P., Vliet, H. (eds.) Software Architecture Knowledge Management, ch. 3, pp. 39–57. Springer, Heidelberg (2009)

17. Krutchen, P.: An ontology of architectural design decisions in software intensive systems. In: 2nd Groningen Workshop Software Variability, October 2004, pp. 54–61 (2004)
18. Nuseibeh, B.: Weaving together requirements and architectures. Computer 34(3), 115–117 (2001)
19. Obbink, H., Kruchten, P., Kozaczynski, W., Hilliard, R., Ran, A., Postema, H., Lutz, D., Kazman, R., Tracz, W., Kahane, E.: Report on software architecture review and assessment, SARA (2002)
20. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Softw. Eng. Notes 17(4), 40–52 (1992)
21. Pohl, K., Sikora, E.: COSMOD-RE: Supporting the co-design of requirements and architectural artifacts. In: IEEE International Conference on Requirements Engineering, pp. 258–261 (2007)
22. Tang, A., Jin, Y., Han, J.: A rationale-based architecture model for design traceability and reasoning. Journal of Systems and Software 80(6), 918–934 (2007)
23. Tang, A.: A rationale-based model for architecture design reasoning. Ph.D. thesis (2007)
24. Tang, A., Han, J., Vasa, R.: Software architecture design reasoning: A case for improved methodology support. IEEE Software 26(2), 43–49 (2009)
25. Tyree, J., Akerman, A.: Architecture decisions: Demystifying architecture. IEEE Software 22(2), 19–27 (2005)
26. van der Ven, J., Jansen, A., Nijhuis, J., Bosch, J.: Design decisions: The bridge between rationale and architecture. In: Dutoit, A.H., McCall, R., Mistrík, I., Paech, B. (eds.) Rationale Management in Software Engineering, ch. 16, pp. 329–348. Springer, Heidelberg (2006)
27. Weinreich, R., Buchgeher, G.: Paving the road for formally defined architecture description in software development. In: 25th ACM Symposium on Applied Computing (SAC), Sierre, Switzerland, March 22-26. ACM, New York (2010)