Combining Architectural Design Decisions and Legacy System Evolution

Sebastian Gerdes[1], Steffen Lehnert[2], and Matthias Riebisch[1]

[1] Universität Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
{gerdes,riebisch}@informatik.uni-hamburg.de
[2] Technische Universität Ilmenau
Ehrenbergstraße 29, 98693 Ilmenau, Germany
steffen.lehnert@tu-ilmenau.de

Abstract. Software development is characterized by ongoing design decisions that must take into account numerous requirements, goals, and constraints. When changing long-living and legacy systems, former decisions have to be considered. In order to minimize the risk of taking wrong or misleading decisions an explicit representation of the relevant aspects is crucial. Architectural decision modeling has proven to be an effective means to represent these aspects, the required knowledge, and properties of a potential solution. However, existing approaches do not sufficiently cover the ongoing evolution of decisions and artifacts. They fail in particular to represent relations to existing systems on a fine-grained level to allow for impact analysis and a later comprehension of decisions. Furthermore, the effort for capturing and modeling of design decisions has to be reduced. In our paper we integrate existing approaches for software architectural design decision making. We extend them by fine-grained traceability to elements of existing systems and explicit means for modeling the evolution of decisions. We show how relevant decisions can easily be identified and developers are supported in decision making.

Keywords: Software architecture, design decision, traceability, evolution, reengineering, legacy software.

1 Introduction

The majority of today's software engineering efforts are spent on continuous and evolutionary development of existing systems [1]. Hence, development faces the ongoing integration, maintenance, and reengineering of existing (legacy) systems. An increasing amount of software is also composed of pre-existing building blocks, such as COTS-components, which therefore represent another type of existing items that have to be considered during design decision making [2].

As software architectures cover many important design decisions, evolutionary development of software systems demands for traceability between decisions and the resulting artifacts to comprehend *who* made *which* decision *when* and *why* [3].

P. Avgeriou and U. Zdun (Eds.): ECSA 2014, LNCS 8627, pp. 50–57, 2014.
© Springer International Publishing Switzerland 2014

Additional traceability between (legacy) decisions is required to enable comprehensive change impact analysis in response to changes. However, this support is not yet sufficiently provided by current research on the documentation and utilization of design decisions. Therefore, our goals are to:

1. Support the evolution of design decisions.
2. Document the origins and potential impacts of design decisions.
3. Establish fine-grained couplings between design decisions, requirements, constraints, and elements of existing systems.
4. Reduce the effort for the modeling of dependencies.

To accomplish these goals, we consolidate the decision models as proposed by Zimmermann [4] and Capilla *et al.* [5]. We augment the resulting decision model with means for fine-grained traceability towards software artifacts which are either impacted by the design decisions or contribute towards them, to help developers understand the implications of their changes. As our main contribution we illustrate how the evolution of every aspect of the decision model is addressed by our approach to allow for a seamless documentation of design decisions. We emphasize how developers are enabled to identify (legacy) decisions relevant to their current tasks and how our approach helps developers to answer the questions arising during software maintenance.

The remainder of this paper is organized as follows. In Section 2 we describe requirements to design decisions, which will be derived from developer's needs represented by use cases. We introduce our revised decision model in Section 3 and explain how our model assists with decision making in Section 4. Related work is discussed based on our requirements in Section 6 and finally Section 7 outlines future research and concludes the paper.

2 Requirements to Architectural Design Decisions

Before analyzing existing works on documenting architectural design decisions we have to define valid criteria for the analysis of the proposed models. These criteria are derived from studies that elicited questions frequently asked by developers during software maintenance and evolution [6,7,8]. These studies revealed general information needs and special demands on software evolution, for which they conducted interviews with developers working in different domains.

2.1 Derived Use Cases and Requirements

In a next step we distilled three use cases from these questions and illustrate how they benefit from explicit design decisions and support developers.

Identifying Relevant Legacy Decisions: If legacy decisions shall support developers with their current tasks, means going beyond simple text searches are required for identifying relevant decisions by querying the set of legacy decisions in a more structured way. The derived use case comprises the efficient access to decisions by limiting the search space, focusing on relevant information, and revealing links to elements of existing systems to support evolution.

Decision Support Based on Legacy Decisions: Once relevant legacy decisions are identified, they must be aligned with the current task to select potentially suitable solutions. Afterwards and in combination with fine-grained traceability towards software artifacts and requirements they enable change impact analysis as one of the key requirements of developers [6]. Legacy decisions and related requirements and constraints reveal why code was implemented in a particular way, which is crucial when trying to understand the rationale behind existing solutions and the evolution of code [6,8]. Historical decisions also expose information about previous issues in terms of constraints, requirements, etc.

Documenting Design Decisions: Developers must be able to populate the decision repository with recent decisions and related information to enable further reuse. This task must be accomplished with as little overhead as possible.

Based on the needs and use cases, we distilled requirements for a metamodel to capture design decisions, which can be summarized as follows.

1. **Explicit support for evolution of decisions and related artifacts:** Considering the evolutionary characteristics of decisions will expose potential pitfalls developers already experienced in the past.
2. **Explicit traceability to related software artifacts:** Fine-grained traceability will show which legacy decision leads to certain artifacts, such as code or models, which will make developers aware of potential impacts of changes.
3. **Explicit traceability to constraints and requirements:** This will reveal the drivers of a decision and the reasons of the developer why code and design are the way they are. They need to be represented as first-class entities.

3 Consolidated Metamodel for Design Decisions

Based on our requirements we propose a consolidated decision model which is displayed by Figure 1 to better capture the evolution of decisions and their relations to other software artifacts, requirements, and constraints.

3.1 Consolidating the Decision Model

The consolidation of the existing decision models is comprised of two steps aimed to increase the applicability of the resulting model. Firstly, we remove several elements from the models which are not necessary for documenting architectural issues and decisions in a real-world context, but complicate the application of the model for developers. Secondly, we revise the attributes of remaining elements and purge those that do not contribute towards the comprehension of decisions.

The first element to be removed is the *ADLevel* as introduced in [4]. There are two reasons, on the one hand its limited benefit when exploring legacy decisions to assist developers to accomplish their task. On the other hand, the boundaries between the different *ADLevels* are quite fluid and most classifications are rather ambiguous, thus misleading developers when documenting issues and decisions.

Furthermore, we identified cases of redundancy in the existing models which should be resolved to streamline the decision model. The first candidates are

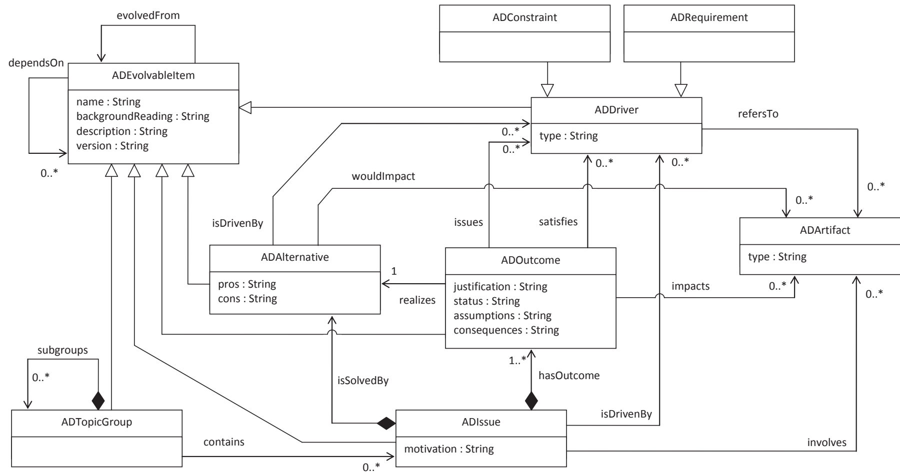

Fig. 1. Our revised decision model represented by a class diagram

ADRequirement, *ADRequirementType*, and *ADRequirementsArtifact* introduced in [5]. We integrated the *ADRequirementType* as an attribute into *ADRequirement*, thereby diminishing the need for a separate class (upper right corner of Figure 1). Likewise, there is little conceptual difference between *ADDesignArtifact* and *ADRequirementsArtifact* as both represent real software artifacts, regardless of whether it is a free text, a use case, etc. By renaming *ADDesignArtifact* into *ADArtifact* and by adding both an attribute *type* to it and a reference from *ADRequirement* towards it we can omit the additional classes.

A similar level of redundancy can be observed in the instances of *ADDesignArtifact*, *ADDesignElementType*, *ADDesignElement*, *ADRuntimeElement*, and *ADRuntimeArtifact*. In this case we also propose to merge the classes into the *ADArtifact* class (right side of Figure 1) for the following reasons: First of all, for the traceability of decisions and issues with design elements the additional layer as introduced by the *ADDesignArtifact* is not required if the granularity of the traceability concept is refined. This will be discussed in detail in Section 3.3. Additionally, the *ADDesignElementType* is dispensable as this information is already encoded in the metamodels of the actual design artifacts.

Finally, we have to reorganize and purge several attributes of the remaining model constituents. To begin with, there is a redundancy in *ADOutcome* class of [4], namely the *candidateAlternatives* attribute which is already encoded by the *isSolvedBy* references of the containing *ADIssue*. This is likely to introduce inconsistencies as architects and developers are forced to link the same entities twice in two different places. There are also various attributes of the *ADIssue* class which turned out to hamper the capturing of issues in practice or were never used, but in turn complicated the representation of recorded information. Therefore, the attributes *phase*, *role*, and *shortName* are removed from our model. Moreover, the attribute *scope* is omitted and instead replaced by more

fine-grained traceability links to software artifacts. This will be explained in Section 3.3 because an issue can encompass more than just one type of artifact.

3.2 Addressing Software Evolution

The next important step towards an enhanced decision model is its support for ongoing development by supporting the evolution of architectural issues, decisions, alternatives, etc. to better reflect real development contexts. Yet current models capture only a small excerpt of an ongoing evolution and thus have to undergo major revisions to support the continuous development.

We observed that every aspect of a decision may evolve over time, including the issues that triggered the decision, potential alternatives and the final outcome of a decision. To address this phenomenon, we introduce the *EvolvableItem* (left upper corner of Figure 1) as a base-class for *ADIssue*, *ADOutcome*, *ADAlternative*, *ADDriver*, and *ADTopicGroup*. By adding the *evolvedFrom* relation to the *EvolvableItem* it is possible to model the ongoing refinement and revision of entities, for example when new constraints were introduced or existing requirements changed. This enables architects and developers to explore and inspect the various influences leading to the current state of the issue, decision, alternatives, and so forth. Moreover, by providing the link back to the previous version of a certain entity developers are able to trace and understand the effects of changes. The mere presence of evolutionary links helps to inform developers about changes of issues etc. which would be lost if all entities would simply be overwritten or replaced. This is especially important for developers joining development at a later stage to focus their attention on recent changes.

3.3 Interweaving Traceability Support and Decision Modeling

Finally, we incorporate an enhanced traceability scheme into our decision model to allow for fine-grained traceability between its constituents. While we keep the relations of Zimmermann and Capilla *et al.*, we add further relations towards software artifacts and provide means for linking dependent entities.

We first introduce *impact*-relations between *ADOutcome* entities and software artifacts represented by the *ADArtifact* entity. Using these relations developers can clearly highlight those artifacts that are impacted by a certain decision, thus assisting with software maintenance. Likewise, potential impacts between *ADAlternatives* and *ADArtifacts* can be expressed as *wouldImpact*-traceability links to signify the consequences of implementing a certain alternative.

Secondly, developers must be able to link architectural issues with the involved software artifacts, as the origin of an issue might result from their interplay. We therefore add the *involves*-relation between *ADIssues* and *ADArtifacts*, which acts as the inverse relation to the aforementioned *impact*-relation.

Moreover, the outcome of a decision (*ADOutcome*) might entail new architectural drivers, like for example when the decision to utilize a SQL-database would impose a new constraint on the storage layer of the software. Hence, the *ADOutcome* is also related through an *issues*-link with the *ADDriver*.

We further extend the scope the *dependsOn*-relation by moving it from the *ADIssue* to the *EvolvableItem*, since our model should support traceability links between different decision entities. As for example, an *ADAlternative* may depends on a previously decided *ADOutcome*, which is not expressible with the current models. Another advantage of this relation is that it provides information about temporal dependencies. It indicates whether a decision has been made before or after another one and would justify it from today's perspective.

4 Supporting Decision Making and Comprehension

The following illustrates how our revised decision model assists developers dealing with the use cases outlined in Section 2.1.

Identifying Relevant Legacy Decisions: As previously stated, simple text searches are not the most feasible way to identify an entry point to documented (legacy) decisions which are related to the current task. Instead, a more structured search approach should be used, both limiting the search space and reducing irrelevant information at the same time. This can be accomplished by exploring the topics in which architectural issues are grouped, allowing a stepwise navigation through the available (legacy) data. We support this by using hierarchic *ADTopicGroup* entities where the lowest level of topic groups finally links towards architectural issues (*ADIssue*). By classifying and refining a current problem, developers can navigate the topic hierarchy to find the topic group(s) containing the most similar issues.

Decision Support Based on Legacy Decisions: Once an initial set of relevant issues and decisions has been identified, the developer must be enabled to decide which of those are most relevant to him. For this purpose three novel aspects of our decision model come into play. Firstly, with the help of our traceability concept the developer can inspect the software artifacts that were impacted by the decisions or which the decisions are based on. Hence, by correlating the impacted artifacts to his current situation he can estimate potential impacts and identify problems. This is further strengthened by the traceability of alternatives and software artifacts to reveal artifacts which would have been impacted. Previous experiences allow to track and understand possible issues and support in balancing current decisions. Secondly, due to our support for linking constraints and requirements with issues, decisions and alternatives, a developer can judge whether similar constraints hold for a project. If so, those (legacy) decisions and alternatives, which do not meet the constraints, can be excluded. This is especially important since similar issues appear in many projects, whereas the outcome is project-specific due to project-specific constraints. Finally, analyzing the "historical" development of a design decision might reveal issues a developer is not yet aware of. By comparing the evolution of a decision, issues which occurred at a certain point in time and might have altered the course of a decision are revealed, thus enabling him to judge the impacts of similar scenarios on the current case. Moreover, it allows to study the refinement of decisions over time which might support in taking the right decision earlier.

5 Evaluation Plans - The CoCoME Case Study

Our evaluation plan is built on the *Common Component Modeling Example* (CoCoME)[1] which was developed to evaluate and compare component-based modeling approaches in a real world context based on the implementation of a trading system for handling supermarket sales and enterprise management. We identified two works that performed various refactorings on CoCoME using different modeling methodologies and also documented their decisions, yet in an unstructured and semi-formal manner [9,10]. Our goal is to apply our model for the documentation of their decisions and to establish the linkage between software artifacts, requirements, and constraints to support software evolution.

6 Related Work

Tang *et al.* [11] proposed AREL as a rationale-based architecture model to document architectural design by means of a UML profile. However, the model lacks dependencies to design elements from design alternatives, which would expose potential impacts. Furthermore, it does not distinguish between constraints and requirements and lacks direct linkage of interdependent decisions. Van Heesch *et al.* [12] proposed a documentation framework consisting of four viewpoints for architectural decisions, which satisfy several stakeholders' concerns but neglect fine-grained traceability to related software artifacts and requirements. Capilla *et al.* [13] developed a web-based approach to capture and manage design decisions, yet it still lacks support for ongoing evolution which is only supplied in a partial manner [14] and fine-grained linking of decisions and software artifacts. Furthermore, Capilla *et al.* [5] extended the metamodel of Zimmermann *et al.* for decision modeling and reuse [4]. Due to their focus on capturing and reusing decisions, the model has various shortcomings in regard to decision evolution and traceability, which were not addressed in a comprehensive manner. They neglect traceability links required for maintenance, i.e. traceability links from issues or alternatives to fine-grained artifacts. Malavolta *et al.* [15] proposed an approach for systematically defining traceability links between decisions to enable "decision impact analysis". However, their linking concept does not provide means to link decisions with artifacts impacted by them. Likewise, linking the artifacts a decision is based on with the actual decision is also not possible either. Che and Perry [16] introduced the Triple View Model (TVM) to manage the documentation and evolution of decisions. The core of TVM is almost identical to Capilla's model, hence both share the same disadvantages. Its major derivation is its support for best practices that are interweaved with the decisions.

7 Conclusion and Future Work

Architectural design decisions provide urgently needed support for evolutionary development, yet current approaches are not fully capable of capturing the evolution of decisions and their fine-grained traceability. Thus we propose a revised

[1] `http://cocome.org/index.htm`

version of an architectural decision model for which we consolidated existing decision models and extended them with comprehensive means for expressing the evolution of their constituents to enable developers to trace the historical development of decisions, their drivers, and their outcomes. We enhanced the obtained decision model with means for fine-grained traceability to enable impact analysis and to further strengthen the integration of decisions when changing long-living software systems during software evolution. Currently, we evaluate our approach in a controlled lab experiment and plan an industrial case study. Further works will focus on the recovery of design decisions and the rationale behind them, as well as on constraints induced by legacy systems to differentiate their impact on design decisions and to underpin their necessity.

References

1. Vliet, H.: Software Engineering: Principles and Practice, 2nd edn. Wiley (2007)
2. Perry, D., Grisham, P.: Architecture and Design Intent in Component & COTS Based Systems. In: ICCBSS 2005, pp. 155–164 (2006)
3. Jansen, A., Bosch, J.: Software Architecture as a Set of Architectural Design Decisions. In: 5th Working Conf. on Software Architecture, pp. 109–120 (2005)
4. Zimmermann, O., Koehler, J., Leymann, F., Polley, R., Schuster, N.: Managing architectural decision models with dependency relations, integrity constraints, and production rules. Journal of Systems and Software 82(8), 1249–1267 (2009)
5. Capilla, R., Zimmermann, O., Zdun, U., Küster, J.M.: An enhanced architectural knowledge metamodel linking architectural design decisions to other artifacts in the software engineering lifecycle. In: Software Architecture, pp. 303–318 (2011)
6. Ko, A.J., DeLine, R., Venolia, G.: Information Needs in Collocated Software Development Teams. In: 29th Intl. Conf. on Software Engineering, pp. 344–353 (2007)
7. Sillito, J., Murphy, G.C., De Volder, K.: Questions programmers ask during software evolution tasks. In: SIGSOFT 2006/FSE-14, pp. 23–33 (2006)
8. Fritz, T., Murphy, G.C.: Using information fragments to answer the questions developers ask. In: 32nd Intl. Conf. on Software Engineering, pp. 175–184 (2010)
9. Knapp, A., Janisch, S., Hennicker, R., Clark, A., Gilmore, S., Hacklinger, F., Baumeister, H., Wirsing, M.: Modelling the CoCoME with the Java/A Component Model. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) Common Component Modeling Example. LNCS, vol. 5153, pp. 207–237. Springer, Heidelberg (2008)
10. Küster, M., Trifu, M.: A case study on co-evolution of software artifacts using integrated views. In: WICSA/ECSA 2012, pp. 124–131 (2012)
11. 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)
12. van Heesch, U., Avgeriou, P., Hilliard, R.: A documentation framework for architecture decisions. Journal of Systems and Software 85(4), 795–820 (2012)
13. Capilla, R., Nava, F., Pérez, S., Dueñas, J.: A web-based tool for managing architectural design decisions. SIGSOFT Softw. Eng. Notes 31(5) (2006)
14. Capilla, R., Nava, F., Dueñas, J.C.: Modeling and Documenting the Evolution of Architectural Design Decisions. In: SHARK/ADI 2007, pp. 9–15 (2007)
15. Malavolta, I., Muccini, H., Smrithi Rekha, V.: Supporting architectural design decisions evolution through model driven engineering. In: Troubitsyna, E.A. (ed.) SERENE 2011. LNCS, vol. 6968, pp. 63–77. Springer, Heidelberg (2011)
16. Che, M., Perry, D.E.: Managing architectural design decisions documentation and evolution. International Journal of Computers 6(2), 137–148 (2012)