# Leveraging Design Decisions in Evolving Systems

Martin Küster, Benjamin Klatt
FZI Forschungszentrum Informatik
Haid-und-Neu-Str. 10-14, 76131 Karlsruhe, Germany
{kuester,klatt}@fzi.de

## 1 Introduction

When a system needs to evolve, it is necessary to not only find code that needs to change, but also to understand why it has been developed the way it was (e.g. which requirements have influenced the software design). Due to cost constraints, missing knowledge, or the initial intention to build a prototype only, it is common practice not to trace requirements to design decisions from the beginning. Existing requirements engineering and documentation approaches provide too generic linking capabilities. Even mature trace frameworks (e.g. EMFTrace[1]) do not provide sufficient support for tracing design decisions and their origins (e.g. requirements). Our approach aims to provide i) models and traces capturing software design decisions, ii) a light-weight process for incrementally building trace information having a minimal impact on design and development efforts, and iii) a view-concept to support developers during future system evolution. To illustrate the application of this concept, we provide an example in the context of migrating several product copies to software product lines and their future evolution.

## 2 Documentation of Design Decisions

Design decisions are major entities in the software development process. Even if software developers often make them implicitly, it is critical to find out i) why a specific design alternative has been chosen, and ii) how a requirement manifests itself in the software. To answer those questions, it is important to prevent software design erosion when the software evolves, and to efficiently perform the necessary changes while knowing the entire context.

Derived from these goals, a design decision links requirements, software entities (e.g. components, interfaces, and classes), and related design decisions (e.g. those made earlier) as shown in Figure 1. Nowadays, mature models and ontologies exist for requirements [6] as well as software entities [5], however the link between these models has not been sufficiently addressed. The trace model that we developed captures design decisions, the references between them, and their related requirements and software entities.

[1] http://sourceforge.net/projects/emftrace/

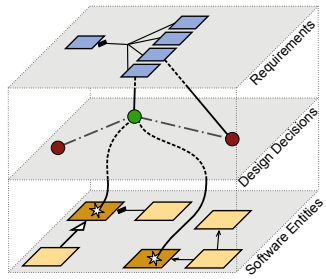


Figure 1: Design decisions linking requirements and software entities

## 3 Design Decision Views for Evolution Support

When a software engineer has to implement a new requirement in an existing system, she might already have an idea which part of the system might be affected. Several existing approaches enable software engineers to locate features that need to be modified in the system. However, the engineer still needs to decide how to modify the system without unintentionally breaking the architecture of the software.

To support her in making a more informed decision, we propose views presenting previously made design decisions related to the software entities she is working on. This includes the latest design decisions (history) as well as links to their (causal) predecessors. In addition to the design decisions, the requirements that caused the design decision can also be made visible.

The challenge of presenting information from different models in one view (requirements, decisions, architecture, design, code) is tackled using a graph-based visualisation. All elements that are related to a focused element can be displayed around it. The potentially large dependency graph can be reduced according to the engineer's current task and the desired kind of relationship (timeline, causal chain, etc.). When navigating through the net of relationships, the graph is auto-layouted using standard layouting algorithms.

## 4 Process to Capture Decisions

The capturing of design decisions faces two major challenges. First, software engineers make implicit

design decisions and are not used to capturing them. Second, there might be resentements against the additional effort. We assume the latter can be offset by the benefits of making the decisions explicit, which in turn saves efforts.

To better support the capturing of design decisions we propose a guided process which makes this intuitve and unconscious. When an engineer makes a design decision to implement a requirement a supporting tool should be aware of her current context (code, architecture) and offer her capabilities to i) describe the decision itself, and ii) link it to the requirements as well as the affected software entities. We propose this as a generic concept which could be further enhanced with domain-specific adaptations. For example, taking domain-specific models into account can be used to present more specific information and to better assess the engineer's current context. Furthermore, integrating the tool into the engineer's individual development environment might enable a better interpretation of her current context and an even more domain-specific presentation and access of the elements. To illustrate a domain-specific application, we give an example from our work on migrating software product copies into common product lines.

## 5 Product Copy Migration Example

Due to organisational and technical constraints, such as costs, time, and evolving domains, software products are often copied and customised without taking concepts such as product lines into account. In [3] we explain how to identify and recommend related variation points in the software. The product line engineer has to make design decisions on i) which variation points to merge, and ii) how to realise their variability. The possible realisation alternatives are influenced by the individual requirements of the product line (e.g. compile-time vs. run-time configurable systems). Those decisions are later used to refactor the product copies to the software product line.

During this process, the necessary information to capture the design decisions is nearly complete: The reason for introducing the variation points is the consolidation of the product copies. The variation point and the corresponding variants are linked to the software entities and the design decision. The rationale for the design decision is represented by the individual requirements of the resulting product line. With a domain-specific tool the engineer can simply accept to capture them in the decision model.

Suppose the product line evolves and a variation point should now be bound during run-time instead of compile-time to enable a multi-tenant system. To realise this requirement, the product line engineer needs to answer the following questions: What architectural elements are touched by the decision? How is the variation point manifested in the design? What was the initial requirement to have compile-time binding? These questions, leaving the scope of a typical view on the system, can be displayed in a view as described in Sec. 3. Exploring the context of the evolution task before actually implementing it will increase effectiveness and efficiency.

## 6 Related Work, Outlook

Bode et. al [1] proposed a quality-focused method for design traceability based on software categories. Various approaches based on semantic wikis have been proposed. They are mostly focused on forward-engineering coming from requirements. Little support for evolution is provided. Preliminary work on the semantics of traces between requirements and architecture has been proposed by Goknil et al. [2]. Our work is similar to Könemann's [4] with respect to explicitly considering design decisions as first-class entities, but Könemann gives no hint on how to exploit this information.

We plan to show the usefulness of the provided views in an empirical study: Two groups of developers will conduct the same evolution task. One group is provided with the respective views gained from the recorded design decisions, and the other is not. To facilitate this, an integration into Eclipse is currently being developed.

In this paper, we outlined how views can support software evolution using documented design decisions. We described a lightweight process to record such design decisions. Giving an example from the area of product-line engineering we showed how the approach can be adapted for a specific domain and the gained knowledge can ease later evolution steps.

## References


[1] S. Bode and M. Riebisch. Tracing Quality-Related Design Decisions in a Category-Driven Software Architecture. In *Software Engineering*, pages 87–97, 2009.

[2] A. Goknil, I. Kurtev, and K. van den Berg. Tool support for generation and validation of traces between requirements and architecture. *Proceedings of the 6th ECMFA Traceability Workshop on - ECMFA-TW '10*, pages 39–46, 2010.

[3] B. Klatt and K. Krogmann. Model-Driven Product Consolidation into Software Product Lines. In *Workshop on Model-Driven and Model-Based Software Modernzation (MMSM'2012)*, Bamberg, 2012.

[4] P. Könemann and O. Zimmermann. Linking design decisions to design models in model-based software development. *Software Architecture*, pages 246–262, 2010.

[5] OMG. Architecture-Driven Modernization : Knowledge Discovery Meta-Model ( KDM ). Technical Report August, OMG, 2011.

[6] A. Tang, P. Liang, V. Clerc, and H. V. Vliet. Traceability in the Co-evolution of Architectural Requirements and Design. In *Relating Software Requirements and Architectures*. Springer Berlin Heidelberg, 2011.