Abstract:
Numerous design decisions are made in model-driven software development which are mostly implicit and not documented properly. Hence, the design knowledge is usually ‘in the designers mind’ and maybe propagated orally, if at all. There exist tools which tackle that problem for architectural decisions which refer to the higher level architecture of the system to develop, but these decisions are not linked to the design models and are only for documentation.

The goal of this project is to contribute concepts and a tool for explicit design decision support in model-driven software development. The contribution is twofold: a technology for creating and storing model differences is developed which will be used for storing model changes for reusable decisions; furthermore, an integration of decision management tools with modeling tools is presented. This allows explicit decision making with tool support and decision reuse which improves the design documentation process and allows sharing and reusing design knowledge.

1 Introduction

Software systems are developed on different levels of abstraction, as the Model-driven Architecture [OMG03] proposes, e.g. starting with a higher level architectural design, continuing with more detailed design models up to code which might be partially generated. Architectural knowledge can be stored and documented in terms of architectural decisions [CND07, TJH07, ZGK+07] which has two major advantages: the design is documented explicitly, and alternative solutions are considered and decisions are made consciously.

Developers also make decisions concerning the design of the system; examples for such decisions are the use of design patterns to solve particular design issues (e.g. from the Gang of Four [GHJV95]) or individual changes with consequences which should be made explicit (e.g. use of polymorphism or the movement of design elements). These decisions usually stay ’in the designers mind’. Such undocumented knowledge is also called tacit knowledge [HAZ07] and might be propagated orally or eventually gets lost.

We believe that it is also worth capturing and storing design decisions on the design-level and not only on the higher architectural level. That design knowledge can then be propagated along and linked to the models. Furthermore, developers benefit from recurring decisions as a form of best practices and patterns.
The complete idea of this project is described in Sect. 2 and consists of two major parts: Sect. 3 discusses how existing decision concepts from the architectural level can be used to support decision making in model-driven development (MDD); Sect. 4 introduces a technology for storing and reusing model differences (similar to a patch for code) for storing model changes resulted from design decisions. Sect. 5 concludes the paper.

2 The Idea

After defining the term design decision, this section discusses the requirements for introducing design decision support in MDD processes.

A design decision in terms of the system’s architecture consists of a design issue or problem, several constraints and assumptions, one or many solutions, and a rationale, amongst others [SLK09]. Harrison et al. distinguish in [HAZ07] between application-generic and application-specific decisions, at which the former are reusable decisions and the latter are only documented for one particular project.

Our focus for design decisions is the design model: a design decision addresses a particular design problem (issue), considering one or many solutions (alternatives), and describes changes in the model according to the selected solution (outcome). This definition is based on the one in [Zim09], in particular concerning the distinction between application-generic parts (issues and alternatives) and application specific parts (outcomes).

Besides the problem that tacit knowledge vanishes eventually, developers mostly use their own well-known solutions instead of considering other, maybe better suited alternatives. An additional problem comes up when the same decision recurs (the same or very similar changes are made to the model), e.g. in similar projects: the same solution is probably realized over and over again which leads to redundant and error-prone work. Further problems are presented in more detail in [Kön09b] which all in all lead to the following goals for this project.

- Document the rationale and consequences for decisions explicitly.
- Link changes in the design model to the outcomes of decisions to make the design knowledge easily accessible at the design model and vice versa.
- Store design decisions in terms of best practices and patterns for reuse for other developers and similar projects.
- Support the developer in making design decisions and considering alternatives; focus on reusable and well-known decisions, e.g. from previous projects.

The idea for realizing these goals is the following. Tool support and several meta models for architectural design decisions already exist, however, they focus on the documentation of a project and rarely on reuse. Thus this project aims to reuse existing decision management tools and a technology is developed for storing design model changes and linking them to the decisions.
3 Design Decision Support in MDD

Some tools already exist for documenting and maintaining decisions, e.g. AREL [TJH07], ADDSS [CND07], and AdkWik [ZGK07]. The latter two allow knowledge reuse in later projects but none of them links the decisions to the design models in the process. This project proposes an integration of such decision management tools with a modeling tool.

Figure 1 shows a modeling tool and a decision management tool at the left-hand and right-hand side, respectively. The center of the figure sketches our extensions for integrating both tools: a decision structure repository stores decision related application-generic design model changes; a decision binding repository contains application-specific bindings which link particular decisions to the respective design model elements; furthermore, an extension to the graphical user interface (design decision view) is added to the modeling tool which addresses the use cases listed in [Kön09b], for instance browsing through made design decisions. Formal meta models for the decision structure and binding as well as the tool interfaces are already defined.

In principle we support any modeling compliant to the Essential Meta Object Facility [OMG06] and any decision management tool which implements the respective interfaces. Our prototype, however, only supports the decision management tool AdkWik and any EMF-based\(^1\) modeling tool so far.

To conclude, we introduce interfaces which the tools must implement – this makes the integration independent of the actual tools. All interfaces are already described in [Kön09a] and a prototype is under development in collaboration with the IBM Research Lab, Zurich.

Related Work

There are several decision management systems available but none of them connects the decisions to the actual models. ADDSS [CND07] and AREL [TJH07], for instance, are

---

\(^1\)Eclipse Modeling Framework: http://www.eclipse.org/modeling/emf/
tools for the documentation of architectural decisions. Moreover, they allow sharing and analyses of the decisions, respectively. AdkWik [ZGK+07], which we use in our prototype, allows to capture, store, and share design knowledge as well; in addition, it supports the user in making and reusing decisions.

Some modeling tools like the IBM Rational Software Modeler\(^2\) and Borland Together\(^3\) provide pattern authoring capabilities similar to the intention of decision structures. However, a meta model for expressing relations between them as well as tool supported guidance is missing.

## 4 Model Differences

For this project, we require to calculate and store arbitrary model changes and apply them to another model as part of a reusable design decision (similar to how a patch is used for code). There are neither technologies nor concepts yet for storing self-contained model differences such that they are reusable, i.e. applicable to other models. This project builds on top of existing technologies for model differencing and provides additional concepts for self-contained model differences and strategies for their creation and application.

Figure 2 illustrates the use of so-called model-independent differences: changes made in a model from one project are extracted and applied to a model from a similar project (which must be capable of applying them). Examples for such changes are design patterns or other recurring changes like refactorings. The individual steps in Fig. 2 are as follows.

\(^2\)Available at: http://www.ibm.com/software/awdtools/modeler/swmodeler

\(^3\)Available at: http://www.borland.com/us/products/together/
1. The *difference calculation* is performed with existing technologies, e.g. with EMF Compare\(^4\), and produces differences which are not applicable to other models.

2. A *transformation* creates self-contained model-independent differences; they are in particular valid without the original models (like a patch).

3. Now the differences can be stored for later reuse, e.g. in a database for best practices.

4. Before the differences can be applied to another model, a so-called *reference resolution* determines *whether* and *where* the changes could be applied.

5. If the previous step was successful, the *changes are applied* to the other model.

The resulting differences from step 2 are *model-independent* in terms of two meanings: first, the differences are self-contained, i.e. independent from the original model they were calculated from (unlike most other differing technologies); second, the differences are also applicable to other models, e.g. from another project – thus, they are independent from the concrete contents of the original models (which is not supported by any other approach so far). The details are described in [Kön09c].

This technology will be used to store and reuse design model changes for reusable design decisions. A prototype is freely available at [http://modeldiff.imm.dtu.dk](http://modeldiff.imm.dtu.dk) and its contribution to the Eclipse project EMF Compare is in progress.

### Related Work

Even though none of the existing model differing concepts have proper support for the creation of self-contained differences, some concepts and goals are very similar to ours. EMF Compare is a generic and extendable comparison framework for all EMF-based models including UML and many more domain specific languages. Unfortunately, the differences are not self-contained and in particular strictly bound to the original model. EPatch\(^5\), an extension for EMF Compare, is a language for describing self-contained patches; but unlike model-independent differences, epatches are not applicable anymore if the models have changed. AMOR [BLSW09] is an adaptable model versioning technique for concurrent versioning systems and defines refactorings as a set of model differences. However, they focus on self-adaptive difference detection and transfer within versioning systems and do not support generic difference creation and transfer.

### 5 Conclusion

This project’s main contribution is to introduce explicit design decision support for MDD processes. To do so, we ’imported’ already existing ideas and concepts from related work of architectural decision management and integrate that with modeling concepts and tools.

\(^4\)Available at [http://www.eclipse.org/modeling/emf/?project=compare](http://www.eclipse.org/modeling/emf/?project=compare)

To operate on models, the second major contribution is explicit and generic patching support for models. It includes a meta model for the definition and the process for the creation and application of self-contained model differences. That part is already finished and performs well for all tested modeling languages, UML amongst others.

A prototype for design decision support is in development, building on top of the framework for model differencing. The evaluation of the concepts and its realization is planned in an industrial project.

References


