Integrating a Decision Management Tool with UML Modeling Tools

Könemann, Patrick

Publication date:
2009

Document Version
Publisher's PDF, also known as Version of record

Integrating a Decision Management Tool with UML Modeling Tools

Patrick Könemann
Technical University of Denmark

Technical University of Denmark

IMM-TECHNICAL REPORT-2009-07

June 11, 2009

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM-TECHNICAL REPORT: ISSN 1601-2321

Integrating a Decision Management Tool with UML Modeling Tools

Patrick Könemann

Technical University of Denmark, Informatics and Mathematical Modelling
pk@imm.dtu.dk

Abstract

Numerous design decisions are made while developing software systems, which influence the architecture of these systems as well as following decisions. A number of decision management tools already exist for capturing, documenting, and maintaining design decisions, but also for guiding developers by proposing potential subsequent design issues. In model-based software development, many decisions directly affect the structural and behavioral models used to describe and develop a software system and its architecture. However, these decisions are typically not connected to the models created during the development process.

In this report, we propose an integration of a decision management and a UML-based modeling tool, based on use cases we distill from a case study: the modeling tool shall show all decisions related to a model and allow its users to extend or update them; the decision management tool shall trigger the modeling tool to realize design decisions in the models. We define tool-independent concepts and architecture building blocks supporting these use cases and present how they can be implemented in the IBM Rational Software Modeler and Architectural Decision Knowledge Wiki. This seamless integration of formerly disconnected tools improves tool usability as well as decision maker productivity.

Keywords. MDA, UML Modeling, Design Decisions, Architectural Decisions.

1 Introduction

Complex software systems are often built using models on different levels of abstraction, as the MDA (Model Driven Architecture) guide proposes [OMG03]. Platform independence and platform specificity are the most crucial abstractions, especially because platform specific models (PSM) can usually be used for automatically generating large parts of the software systems. UML (Unified Modeling Language) [OMG07b] is just one modeling language for describing these models e.g. using class and component diagrams for structural, and activity diagrams and state charts for behavioural aspects. Current decision support systems like the *Architectural Decision Knowledge Wiki* (AdKWik) [ZKL+09] and the *Architectural Design Decision Support System* (ADDSS) [CND07] help to document, to understand, and to build several genres of software systems. Considering model-based software processes, however, they do not yet involve the design models but are rather designed for documentation of the software system and for guiding developers by proposing potential and possibly mandatory subsequent design issues.

Research about tool support for software architects discusses viewpoints, separation of concerns, and the role of different stakeholders, amongst others. The IEEE 1471 Standard [IEE00] explains conceptually, how an architectural description of a system can be developed with respect to the last-mentioned aspects. However, it does not give much information how architectural knowledge and decisions can be connected to the architectural artifacts. In [Kru95], [Kru95] explains different viewpoints on the system, each having a particular purpose for particular concerns. But design decisions are still implicit and should be made through several viewpoints. The integration we propose in here can be seen as part of a viewpoint for design or architectural decisions, used by both modelers and architects.

The report first presents a case study about a simple MDA process to point out the use and re-use of design decisions. The process deals with creating a web application, but decision

management is also applicable to any other application genre. This case study focuses on bringing models and their changes together with design decisions.

Then we distill use cases as requirements for the integration of a design decision management and a modeling tool (although we refer to UML as modeling language, any other language should be feasible as well): as a modeling tool we use the Rational Software Modeler[1] for developing UML models, because it relies on the freely available and widely used UML2 project[2]; as a design decision management tool we use the AdKWik for organizing and maintaining design decisions during the process, because of its sophisticated meta model for decisions. Use cases for both ends are listed to cover the main functionality of the interaction between both kinds of tools. Each use case contains a description of the main scenarios and a possible realization.

The last part presents an approach for integrating both tools such that, on the one hand, made decisions can be propagated and applied to the models, and, on the other hand, that the modeling tool can make use of design decisions. That is, it shows information about made design decisions and provides functions for making new decisions. For this purpose we propose the required interfaces for both tools as well as the connecting functionality.

2 Background

This section gives useful background information by explaining important terms, the overall development process we refer to, as well as some general requirements and assumptions.

2.1 Terms

The following terms are important for further understanding. Concerning design decisions, we distinguish between the *issue*, *alternative*, and *outcome* of a design decision as explained in [ZGK+07, ZKL+09]:

Design Issue: A design issue is a particular description of a design problem which can be described in a formal model in a decision tool; it contains information about the decision drivers, a set of solutions, called *alternatives*, and a scope, which gives information which elements in the *design model* are relevant for this issue.

Alternative: An alternative is a particular solution to a design issue; it contains information about advantages and disadvantages of the solution, and also a scope which is typically the same or a subset of the scope of the respective issue.

Outcome: An outcome is a made decision for a design issue; it contains a justification and points to the selected alternative; it also refers to the participating design model elements according to a *decision structure*.

Design Model: A formal model used for the development of a software system, in our case we refer to UML models; it usually covers the structure and sometimes also the behavior of the software systems and is typically used for code generation (cf. PIM and PSM in the MDA guide [OMG03]).

Decision Structure: Some design decisions result in design model changes which can be described by a decision structure; a decision structure is a pattern which might appear in several model – hence, it is independent from particular design models.

Decision Binding: A decision binding is a relation between an outcome, a decision structure, and a design model; that is, it defines how a decision structure for a particular outcome is applied to a design model; an outcome / a design model element is *bound*, if it takes part in a decision binding.

[1] http://www.ibm.com/developerworks/rational/products/rsm/

[2] An EMF-based (Eclipse Modeling Framework) implementation of the UML 2 metamodel: http://www.eclipse.org/uml2

Both, issues and alternatives, are independent from any project – they statically describe the problem and possible solutions. An outcome, on the other hand, is the result of a made decision in a particular project. However, for simplicity we consider only issues in the case study which are relevant for the considered project.

2.2 The Development Process

We refer to the MDA, i.e. models are used on different levels of abstraction in order to describe the software system in the different phases of the process. We distinguish between a model (or data model, e.g. a logical data model) and diagrams, which only *visualize* all or some elements of a model. A model may have many diagrams, but a diagram always refers to one model. If we talk about a diagram without mentioning a model, we assume that an underlying model exists. The following is an idealized process for the software design which is divided into three refinement steps:

1. **Solution Outline (analysis phase)**
   The system outline may contain some (usually informal) models and documents:
   - SCD (*System Context Diagram*): shows the system to build and all external systems it interacts with.
   - AOD (*Architectural Overview Diagram*): shows the main components of the system to build.
   - NFR (*Non-functional Requirements*): Define some non-functional requirements for the system, e.g. interoperability, maintainability, or scalability.
2. **Macro Design (step between analysis and platform independent design phase)**
   The macro design contains two different models:
   (a) LCD (*Logical Component Diagram*): shows the logical structure of system from two different points of view. The *static* part is shown with CRD (*Component Relationship Diagrams*, e.g. UML component- or class-diagrams), and the *dynamic* part is shown with CID (*Component Interaction Diagrams*, e.g. UML sequence-, activity-diagrams, or state-charts).
   (b) OM (*Operational Model*): shows the physical components of the system, e.g. physical servers and network infrastructure.
3. **Micro Design (platform specific design)**
   The micro design roughly contains the same as the macro design, it is just more fine-grained. E.g. it usually only uses class-diagrams for refining CRD, and not component diagrams any more.

2.3 Assumptions

The following list gives some important assumptions and emphasizes some facts concerning the general development process as well as the case study:

- Enterprise applications form just one of many application genres. We use the design and development of a web application in the case study for illustration purposes – our concepts and their implementation must work for any architectural style.
- Logical component model and physical operational model are two important architectural artifacts, typically a combination of one or more models and text. The model part must be supported by our solution; support for or integration with text capturing facilities is needed in practice, but future work.
- Regarding the design model, we only focus on *structural* models and diagrams.

2.4 Requirements

The following requirements pertain to the case study and to our proposed solution respectively.

- The solution must work with UML profiles (stereotypes in particular) if available for certain application genre/architectural style; however, it should not depend on such profile, it may not exist or may not be used by the developers in the project.
- We have to support architects when they apply their techniques to master complexity, e.g. functional decomposition (zoom in), layering, and refinement levels.
- Multiple patterns might appear in one architectural element. Hence, all relevant design decisions for that element must be visible in the models.
- It must be possible to record decisions made as outcomes from model changes, either automatically or manually (decision mining).

2.5 Decisions

Before presenting the case study we already made some decisions concerning our concepts and solution. Most of them do not restrict the usage of our solution but are just a matter of simplification.

- We treat a high level system context diagram as a special form of UML component or class diagram.
- We represent the logical component model artifact as a single model (there might be multiple diagrams).
- Decision-related text has to be added to the design model elements, if they are involved in design decisions; furthermore decision-related information should be shown to the user in the modeling tool.

3 Case Study

The goal is to model and develop a *web application* which presents customer contracts to agents, provides forms to modify the contracts, and finally stores the modified contracts. The data is stored in a relational database. So the overall genre is EA (*Enterprise Applications*). The remainder of this section designs the system according to the process described in the introduction. Please note that the decisions are written rather informally in the following format, a formal definition of decisions (especially concerning model changes) is discussed later in Sect. 5.

> **Decision 0:** ‹**name of the design issue**›
> ‹*detailed description of the problem*›
> ‹**outcome (selected alternative)**› – ‹description, justification, and further information about the outcome›

So the *issue* or *problem* the decision refers to is named and described in the first half, whereas the last part combines the outcome and the according alternative. Be aware that there might be several alternatives and many more properties for each issue, however, a simplified form will suffice for this case study.

3.1 Designing the System Outline

We start designing the system outline by making a first decision which concerns the overall architecture:

Decision 1: Architectural Style
Selection of the overall architectural style
Layered architecture – Composing the system in different layers separates responsibilities. A simple form is a 3-layer-architecture as presented by Fowler [Fow02], which includes a Presentation-, Domain-, and Data-Access-Layer.

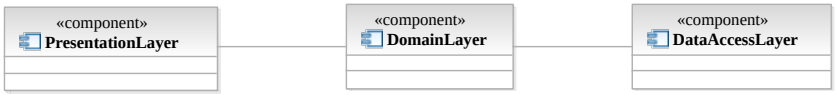

Figure 1: The different layers of the web application

Figure 1 shows the three layers as a UML component diagram. The responsibilities are defined as follows. The *presentation layer* prepares the data for the agent, interacts with the agent, and transfers the data to the domain layer. The *domain layer* contains the domain specific logic to manipulate the data. The *data access layer* just handles the access to the underlying database to retrieve and store the data.

After the decision is made, the three components build a first model for the system. Since it is a creational step, we call it *creational decision making*. Decision 1 enables the next issue which already pertains to the macro design.

3.2 Creating the Macro Design

There is no sharp separation between the different design phases. However, the next decision defines the internal structure of a higher level component; so this justifies the step into the macro design phase.

Decision 2: Presentation Layer Pattern
Internal representation of the presentation layer
MVC pattern – The Model-View-Controller pattern [Fow02] contains three components with different purposes. The view is responsible for presenting the data and interacting with the user, the model for holding the data, and the controller maintains and controls all changes done to the model.

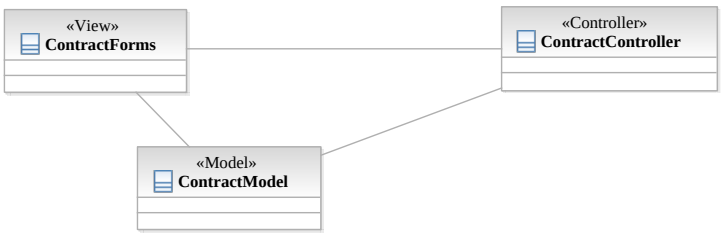

Figure 2: Components of the Model-View-Controller pattern

This decision *decomposes* the presentation layer component into several subcomponents. Figure 2 shows these components as a UML class diagram with names according to the web application

(prefixed with "*Contract*"). In order to mark the different roles, the pattern makes use of stereotypes. The new components will be added to the existing model, so we call it *extensional decision making*. However, to visualize the components, a new diagram is created. This decision again influences the next issue.

> **Decision 3: Session Awareness**
> *In a web application the presentation layer often needs to maintain a session for a client.*
> **Yes** – The controller needs to store the state of the client interaction if more than one interaction step is performed or multiple agents work with the application simultaneously.

Decision 3 does not change anything in the model, however, it implies the following issue. This is why we call it *relational decision making*, i.e. its intention is to influence related decisions.

> **Decision 4: Session Management**
> *Specifies the strategy for managing the session as proposed in [Fow02]*
> **Server Session State** – The server is responsible for the session state and thus requires a session object to store it. In combination with a Model View Controller pattern (decision 2), the controller is the responsible component.

Let us assume that due to the work of another developer on the application, a session manager component already exists in the model. So we do not want to create a new session manager but rather re-use the existing one. Again, this is *extensional decision making*, because the existing model (and even the existing diagram) is extended.

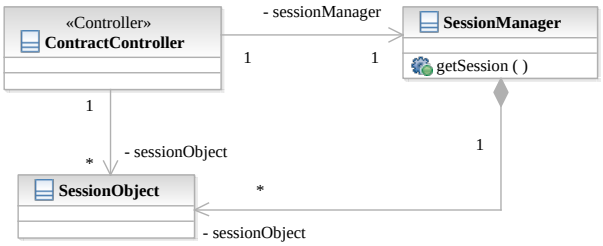

Figure 3: Session management added to the presentation layer

As shown in Fig. 3, a session object and some references were added, all other elements are re-used. In order to express the re-use of existing elements for decision making, we use the term *binding*, i.e. the session manager is *bound* to this particular decision. The following issue is again influenced by this decision.

> **Decision 5: Session Persistence**
> *Is the session state persistent?*
> **Yes** – A persistent session state is stored, e.g. in a database. This information is usually stored as an attribute for the session object, since other session objects might have a different persistent setting.

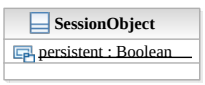

Figure 4: An attribute defining the persistence of the session object

Decision 5 introduces an attribute in one of the components in the design model, as shown in Fig. 4. This is why we call this *property decision making*. In the example, the new attribute is a constant defining the session persistence to be always true.

While working with the models, a designer realizes that the component *model* should rather be placed in the domain layer instead of the presentation layer. So he moves it into the domain layer and decided to introduce a new design decision for that change:

> **Decision 6: Generic movement**
> *The selected set of components should be moved to another location in the model.*
> **Move to another package** – Although the model is part of the MVC pattern in the presentation layer, it should rather be placed in the domain layer because all domain-specific information should be located there.

Decision 6 is a newly created decision and thus needs to be stored next to the other decisions and marked as *decided*. The procedure is outlined as follows. First, the change (in this case the movement) is performed by the designer. Second, the designer uses the modeling tool to select the change and to create a new design issue with a new alternative. Third, the tool requests some general information from the user, e.g. a name, description, the structural change, as well as concrete information about the decision made, e.g. a justification, pros, and cons. Fourth, the tool stores the decision in a catalogue of all design decisions and binds the relevant model elements to the outcome of the new decision.

This movement has obviously violated the changes made in decision 2, which requires all components of the MVC pattern to be placed in the same package. Hence the tool should alert the user. Let us further assume that an issue for a replacement of a component called *component replacement* already exists; however, the existing alternatives are not adequate for a placeholder of the *model* in the presentation layer. Thus the user changes the model in such a way that a wrapper component is added to the presentation layer which delegates its responsibilities to the moved component *model*. Similar as before, the user selects that change, but this time a new alternative is creates for the existing issue:

> **Decision 7: Component placeholder**
> *A set of components is replaced by another component which represents the former.*
> **Wrapper** – In contrast to the proxy (another alternative), a wrapper is more general and does not yet fix the implementation method. I.e. a decorator or adapter pattern might be appropriate as well (see also [GHJV95]).

After decisions 6 and 7 are made, the relevant model elements are shown in Fig. 5; we did not make any decision concerning the domain and data access layer, so we omit their details. The model structure is based on packages, i.e. the model elements for each layer are contained in the respective packages.

Variants

It is worth discussing slight variants for the decisions 6 and 7, because they illustrate interesting scenarios concerning the integration of decision management and UML modeling. Assuming the general scenario that the designer realizes (while working with the modeling tool) that a design decision is missing, its creation may be performed in various ways. Table 1 summarizes the relevant cases[3]: either the issue is new and has to be created (variant 1), or the issue is already known and only the actual change in the model is new (variants 2 and 3). Decision 6 is an example for the first, decision 7 for the third variant[4].

Summary

The solution outline covered the basic platform independent aspects of the web application. Although there are many more design issues, the presented ones are sufficient for the case study to illustrate our work. In the next step, we refine the design to be platform specific, i.e. we propose a transformation to the micro design.

[3] There are more combination possible; however, we do not see any reasonable use for other combinations.
[4] It is yet subject of further investigation how the user interaction will be realized.

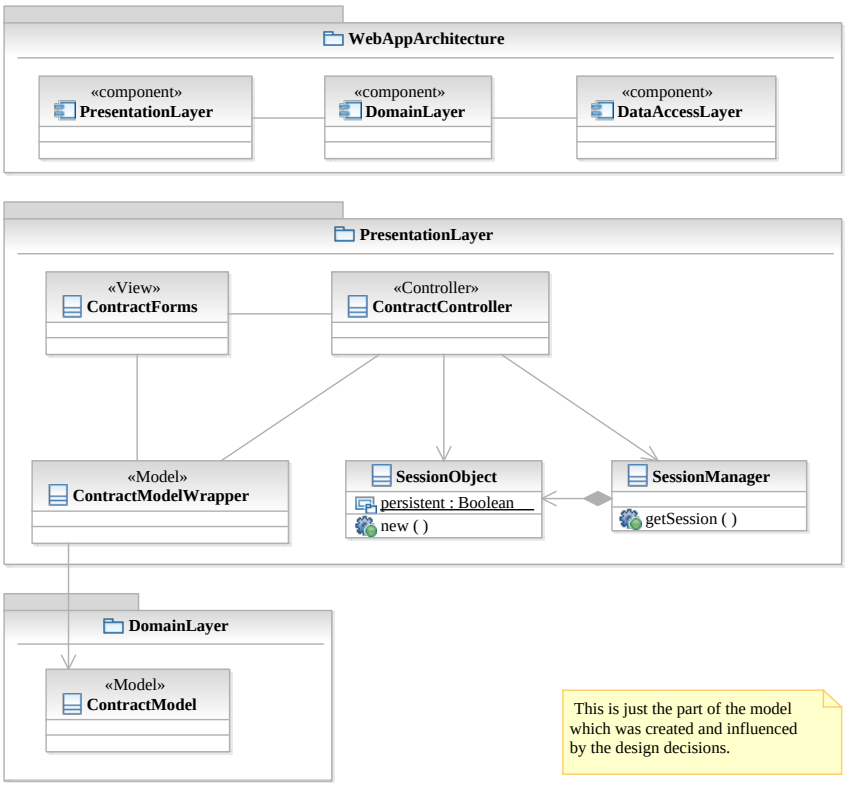

Figure 5: Overview of macro design

Variant	Issue	Alternative	Outcome	Model
1	create new	create new	create new	change
2	select existing	create new	create new	change
3	select existing	select existing	create new	change

Table 1: Variants for creating a new design decision

3.3 Creating the Micro Design

In order to transform the platform independent macro design to an initial platform specific design, we need to make several decisions concerning technologies.

Decision 8: Programming language (TSD[5])
Which programming language will be used to build the system?
Java – Java is available on many platforms.

Decision 9: Java Version (TPD[6])
Old Java version 1.4 for compatibility or a new version: 5 or 6.
Java 6 – The system will benefit from features of the newer Java versions.

Decision 10: Communication technology (TSD)
The technology used for communications between the system (in particular the presentation layer) and the client(s).
Web via HTTP – The communication will be via the Internet, in particular via HTTP.

[5]Technology Selection Decision, defined in [ZKL+09]
[6]Technology Profile Decision, defined in [ZKL+09]

Decision 11: Communication technology profile (TPD)
Which version and other details are used?
HTTP 1.0 and HTTPS 1.0 – The system will use HTTP in version 1.0 and secure HTTP for the transfer of sensitive information.

Decision 12: Webserver selection (ASD[7])
Which webserver product will be used?
Apache – The Apache webserver is a powerful and flexible open source webserver.

Decision 13: Webserver configuration (ACD[8])
How is the webserver configured?
Virtual server – To provide maximum flexibility, the server is configured as a virtual server in the apache configuration.

If these decisions are made, a transformation from the platform independent to the platform specific design can be performed. Here we assume that we use a rule-based transformation language such as QVT (Query/View/Transformation) [OMG07a] or TGG (Triple Graph Grammars) [Sch94]. To do so, we need to adjust the transformation rules according to the previous technology-based decisions.

For a clear separation between the platform independent and platform specific design, we introduce a new model. The goal is to use a traceable transformation, i.e. the relation between the source and the target model elements are stored. Figure 6 shows the package of the presentation layer in the macro design on the left- and transformed to the micro design on the right-hand side. *SessionManager* was replaced with *ApacheHttpSessionManager*, *SessionObject* with *ApacheSessionObject*, etc. In between there are some trace nodes to keep track of the relations between the micro and the macro design.

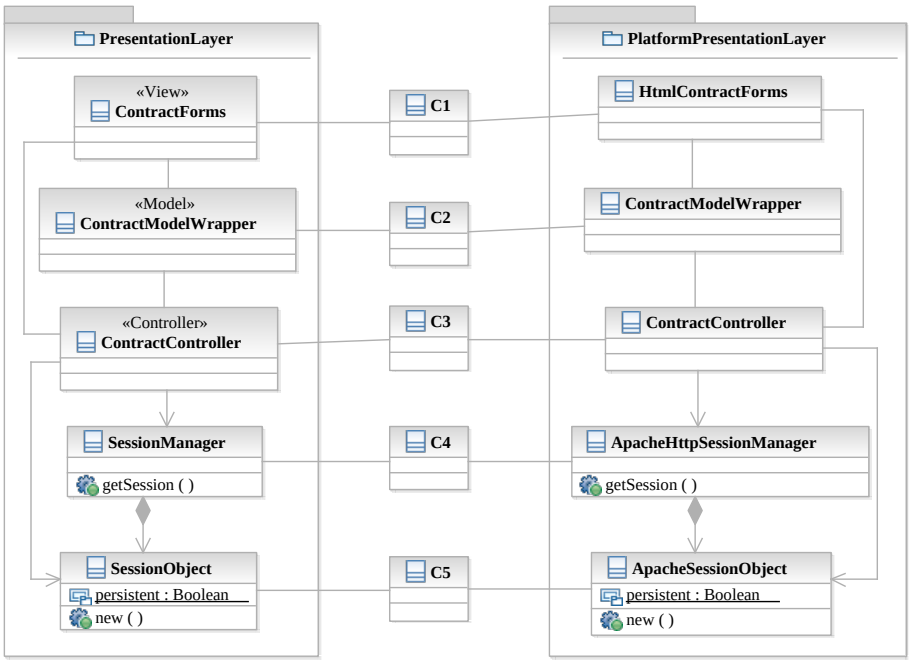

Figure 6: The macro design of the presentation layer transformed to the micro design

[7] Asset Selection Decision, defined in [ZKL+09]
[8] Asset Configuration Decision, defined in [ZKL+09]

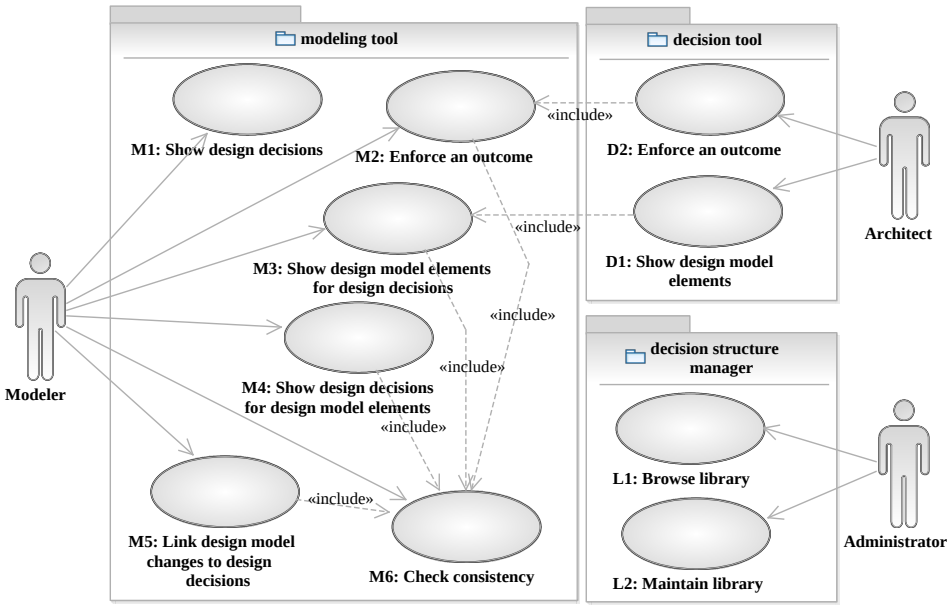

Figure 7: Overview of and relation between all use cases

The same techniques used in Sect. 3.2 can now be used to refine and extend the platform specific models. As a final step, the platform specific models could be used for generating parts of the code. However, this is not in scope of this case study.

4 Requirements

Based on the case study, we now distill the main use cases as the requirements for combining UML modeling with design decisions. They are split into three parts: use cases for the UML modeling tool, for the decision tool, and for a component called *Design Structure Manager* for maintaining the library which connects the UML models and the decision model. The library will be explained later in more detail, for now it is sufficient to know that it contains the information of how decisions and design model elements are connected. Three actors are involved in the use cases: the *modeler* who works with the modeling tool, the *architect* who works with the decision tool, and the *administrator* who has direct access to the library. An overview and the relation between the different use cases is given in Fig. 7. The use cases are presented in the following schema:

Use Case <**use identifier**>**:** <**use case name**>
<detailed description of the use case>
⇒ <*possible realization*>

4.1 Use Cases for the Modeling Tool

This section contains a list of use case scenarios concerning the modeling tool which is mainly used by modelers although architects might also use it, especially in the macro design phase. Figure 8 shows a snapshot of the user interface of the Rational Software Modeler to which the scenarios refer. It is just an exemplary layout to illustrate the ideas: the model explorer allows one to browse through the model whereas the model editor can be used to create and modify the model through diagrams; the parts in italics are our proposed GUI extensions and described in the following use

cases.

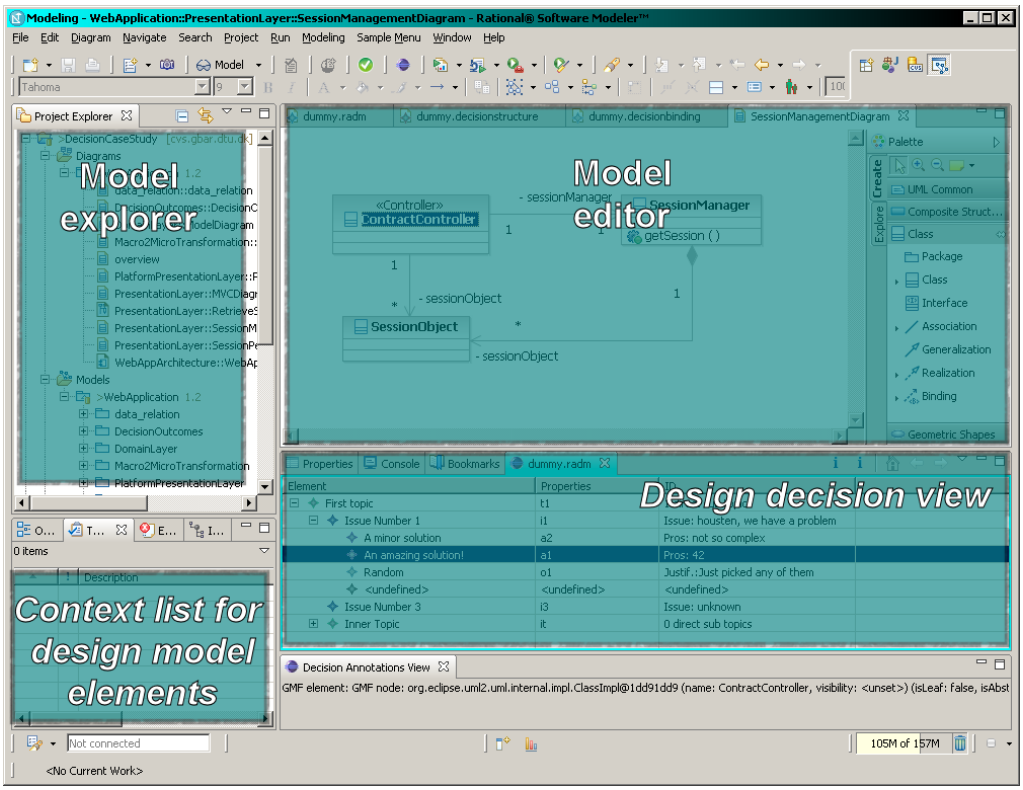

Figure 8: Snapshot of the GUI of the Rational Software Modeler

Use Case M1: Show design decisions
All project related design issues are browsable (read-only) from within the UML tool although they are technically located in the decision tool. The use case contains two different scenarios:

1. All design issues and alternatives are shown.
2. All outcomes can be shown in addition to issues and alternatives.

⇒ *A new view seems to be suitable (*design decision view *in Fig. 8), which lazily loads and presents all issues, alternatives, and outcomes in a tree-based table.*

Use Case M2: Enforce an outcome
An alternative is selected for a decision, hence, an outcome is enforced. That means that the UML model needs to be changed if the selected alternative requires it. The action can be triggered either from the view described in use case M1 or by the decision tool (use case D2) and consists of two steps:

1. Selecting the involved design model elements.
2. Changing the design model.

After the action succeeded, a binding exists for traceability reasons which links the outcome to the respective design model elements. The binding is however technically located and maintained in the library (cf. Sect. 4.3). The outcome status is set to *decided*.

⇒ *A wizard seems to be suitable for the required user-interaction:*

- *The design model elements, which should be modified, have to be selected be the user.*
- *Some decisions need parameters which define* how *the design model will be modified.*

Use Case M3: Show design model elements for design decisions
The selection of decision elements provided by use case M1 results in a context-sensitive list of design model elements:

1. *Issue:* Its scope is listed, i.e. the set of design model elements which are relevant for this particular issue.
2. *Alternative:* Its scope is listed, i.e. the set of design model elements which are relevant for this particular alternative; note that it might also differ from other alternatives of the same issue.
3. *Outcome:* All bound design model elements are listed.

⇒ *A context-sensitive list view for design model elements seems to be suitable which might also contain information about the respective roles of the elements. A subsequent action might locate the elements in the model explorer.*

Use Case M4: Show design decisions for design model elements
A selection of design model elements in the model explorer or editor shows the following information concerning design decisions:

- Show all outcomes in which all design model elements of the selection are bound.
- All open issues in the project can be queried for which the selected design model elements are covered by the scope. Since this might be an extensive operation, it must be triggered explicitly.

⇒ *This can probably be realized as selection listeners in the model explorer, model editor, and design issue view.*

Use Case M5: Link design model changes to design decisions
The user has the opportunity, after modifying the design model, to document the changes and make them reusable as part of a design decision. To this end, the decision structure is computed and attached to a new or existing design issue:

- The decision structure can be attached to an existing alternative of an existing design issue.
- The decision structure can be attached to a new alternative (which needs to be created) of an existing design issue.
- The decision structure can be attached to a new alternative and a new design issue (both need to be created).

Afterwards, the respective design model elements are bound to the newly created outcome (cf. use case M2).

⇒ *A wizard seems to be most suitable to handle the user-interaction in subsequents steps:*

1. *The decision structure has to be generated and its parameters have to be defined.*
2. *An issue must be selected or created.*
3. *An alternative must be selected or created.*
4. *An outcome must be created.*

Use Case M6: Check consistency
A consistency check validates the bindings between design model elements and outcomes. The result will then be presented to the user. The validation criteria are the following:

- All bound design model elements must exist in the design model (e.g. classes and assiciations).
- All constraints of the decision structure must prevail (e.g. concerning attribute names or stereotypes[9]).
- All bound outcomes must exist in the decision model and the decision tool allows its realization[10].

⇒ *A validation action can be provided e.g. in the toolbar. In addition, this function shall be reused by use cases M1–M5.*

4.2 Use cases for the decision tool

The design decision tool is used by architects and to some extend also by modelers to maintain all design decisions made in the project. Figure 9 shows a snapshot of the AdKWik, again to examplarily illustrate our ideas: the design issue browser allows one to browse through all available design issues of the current project; the description and a list of alternatives of a selected issue are shown in the center; in addition, relations between issues and further information are shown; the part in italics is our proposed GUI extension as explained in the following use cases.

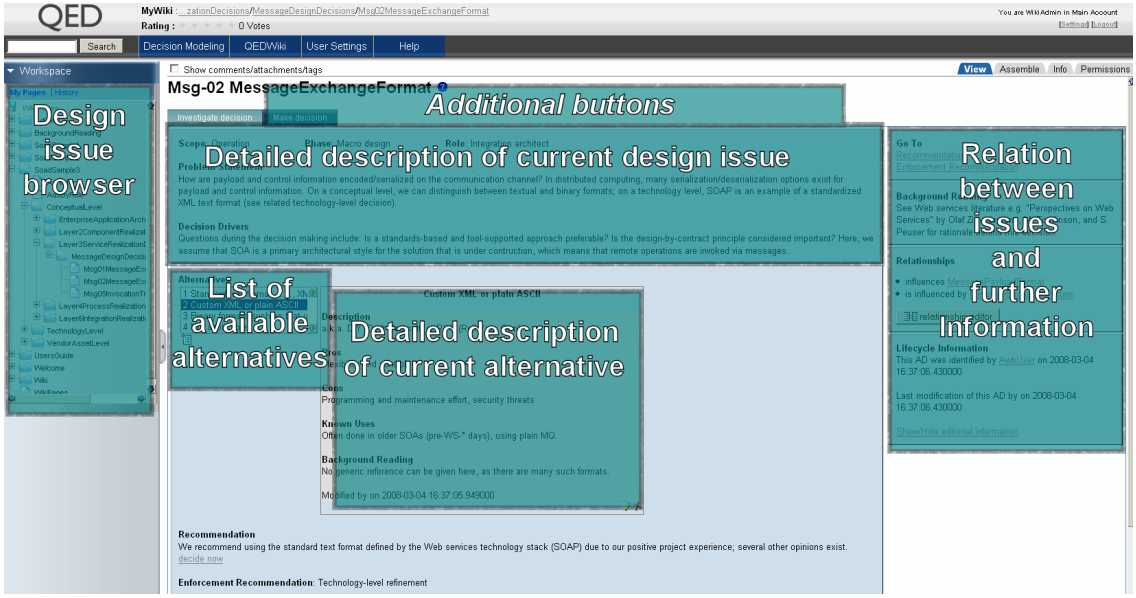

Figure 9: Snapshot of the GUI of the AdKWik

Use Case D1: Show design model elements
There are design model elements and decision structures related to design decisions:

- Show all design model elements bound to a particular outcome, i.e. all design model elements which are involved in a decision.
- Show all design model elements which are relevant for a selected issue or alternative (cf. scope listing in use case M3).

[9] The exact form and properties of decision structures is future work.
[10] We introduce a status for that; this is discussed later in Sect. 5.2.

- Show the decision structure temporarily applied to the design model; this allows the designer to see the resulting changes without already enforcing the decision.

⇒ *The decision tool has to provide actions for the described cases, e.g. with buttons, if the respective elements are selected. Then it delegates the requests to the modeling tool (use case M3 for the first two scenarios and the second step of use case M2 for the third scenario).*

Use Case D2: Enforce an outcome
If a decision is made, i.e. an alternative of an issue is chosen for an outcome, the status of the outcome changes to *decided* and the modeling tool is requested to the the models accordingly (use case M2). The modeling tool is then responsible for setting the outcome status to *decided* after changing the design models.

⇒ *Again, a button in the decision tool would probably suffice to delegate the request to the modeling tool.*

Although use cases D1 and D2 are relevant for an architect, they technically delegate the responsibilities to the modeling tool because the described functionality deals with the design models. Further research will determine whether is would be practical to return information from the decision structure library back to the decision tool for presenting it to the architect.

4.3 Use cases for the library of decision structures

The complete list of use cases for the library is subject to ongoing work. The basic two use cases for administrators, however, are the following.

Use Case L2: Browse library
List the decision structures for alternatives (project independent) and show the bindings for particular decision outcomes.

⇒ *Probably the library just contains an API to access the data without having a separate GUI for this use case.*

Use Case L2: Maintain library
The user must be able to manually define, modify, or delete decision structures and decision bindings.

⇒ *Again, an API would probably be sufficient for this component.*

4.4 Summary

We distilled the main use cases for decision support from the case study in Sect. 3, categorized by the individual components. The most important and detailed use cases belong to the modeling tool, because it is the primary place for the integration of both tools. The use cases concerning the decision structure library are important for the access of the decision structures, however, it needs further investigation whether a GUI is required for it.

One important conclusion is the extension of the decision tool. Since both use cases can also be accessed directly from within the modeling tool, the GUI extension in the decision tool is optional. However, the decision tool must provide an interface to allow access to the design decision model.

The remainder of the report deals with a possible architecture for realizing the use cases.

5 Conceptual architecture

This section proposes an architecture of how the two kinds of tools, a decision management and a UML modeling tool, can be integrated. For this solution we assume that both tools are

extendable concerning data access and interaction. In particular, we define interfaces for both tools which allow appropriate interactions. First we present an overall architecture of all participated components in Sect. 5.1, whereas the remainder explains each role in more detail. During the development of this tool architecture, we made several design decisions:

- We use UML as a modeling language as it is commonly used in the domain of software engineering; however, our work should also work with any other kind of modeling language.
- The decision model is (and remains) independent from the design model, and vice versa.
- Some new components will be plugged into the modelling tool to extend its GUI.

5.1 Overview

The intention of our architecture is to keep both tools, and in particular the design model as much as the decision model, independent from each other. However, in order to combine design decisions and design models we require several interfaces by these tools. They are shown in Fig. 10, along with two additional components: the *DecisionStructureManager* is responsible for coordinating all communication between both tools and can be seen as a facade for the modeling tool and the library; the *DecisionStructureLibrary* is responsible for storing decision structures as well as decision bindings.

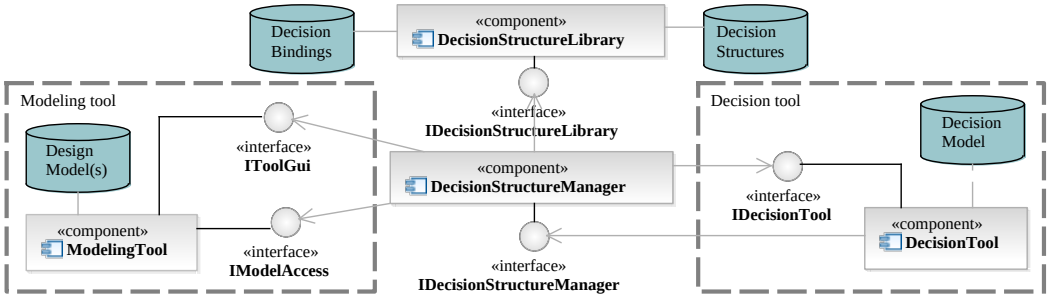

Figure 10: The overall architecture for combining a decision and a modeling tool

The use cases imply two directions of component communication from the user's perspective. First, the architect may trigger an action in the decision tool, e.g. making a particular decision. Then the tool needs to notify the decision structure manager to apply that decision, which means that it has to retrieve the according decision structure from the library and to modify the design models accordingly. Second, the modeler uses GUI extensions in the modeling tool to request information for a particular design decision. Then the information has to be collected from both the library and the decision tool.

Figure 11 shows the implicit relation between decision outcomes of decisions 2, 3, and 4 of the case study (cf. pp. 5), and the involved design model elements (dashed lines). There is in particular no connection between outcomes and design model elements in the current tools, instead the relations are maintained manually.

Fig. 12 sketches the same relation between the different models as Fig. 11, this time including decision structures and the explicit binding: the design model at the bottom shows the presentation layer from the case study (same as in Fig. 11); the decision model at the top shows the outcomes of the considered decisions and the selected alternatives (simplified compared to Fig. 11); the decision structures belong to the respective alternatives and define the structural changes of the design model like a pattern; there is one decision binding for each outcome which maps the decisions structures to the associated elements in the design model.

Here we can make some important observations: the relations between design decisions are only stored in the decision model; the decision model *does not refer* and does not know about the decision structures; the design model *does not refer* and does not know about the decision

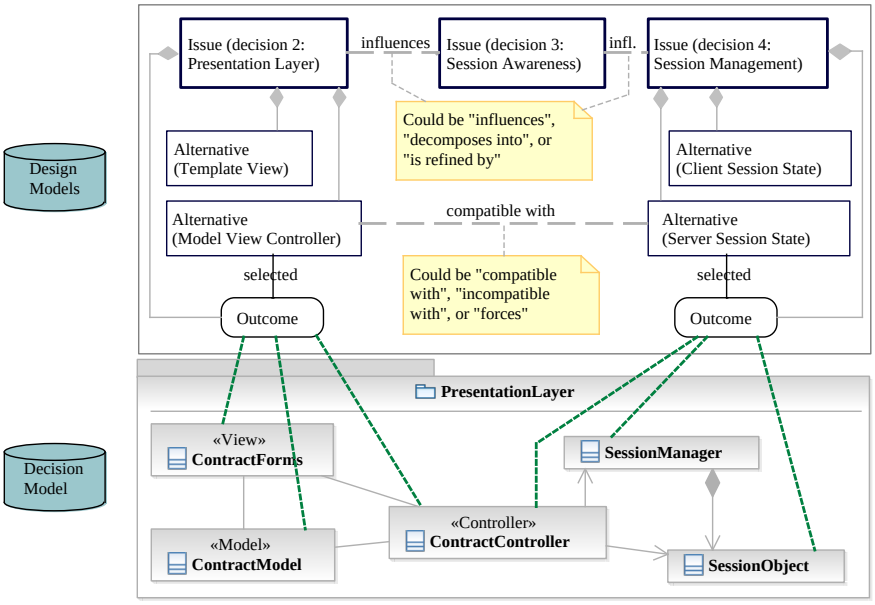

Figure 11: Implicit relation between the design and the decision models (relations between issues and alternatives are defined in [ZKL+09])

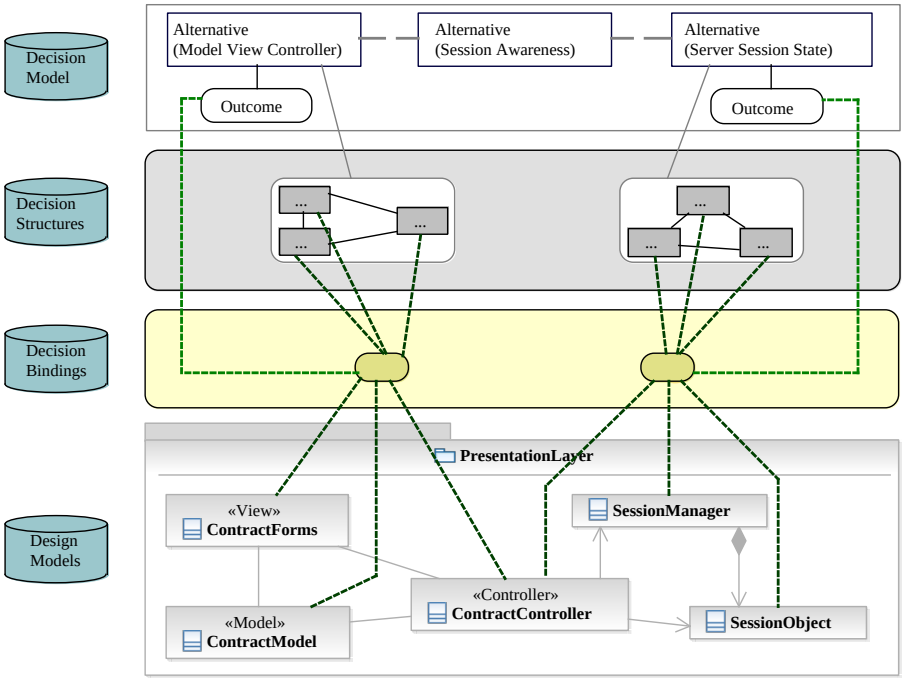

Figure 12: Explicit binding between the design and the decision models

bindings. This ensures the independence of the models of both tools. Next we have a more detailed look at the individual components of Fig. 10, their roles and responsibilities.

5.2 Decision tool

The role of a decision tool is to create and maintain decisions in a process. This includes the definition of decisions and their relations, their consequences, and further informal information. With regard to a concrete project with one or more design models, the decision tool allows the user to make and document design decisions; some of them might be related to concrete design model elements (e.g. decisions 2 and 4), some are not (e.g. decision 3). Concerning the design models, the decision tool has the following responsibilities:

- If the user makes a decision, the decision tool has to notify the decision structure manager to update the design model and bindings, if necessary.
- The decision tool has to provide sufficient decision related information for particular design model elements, which will be shown in the modeling tool.

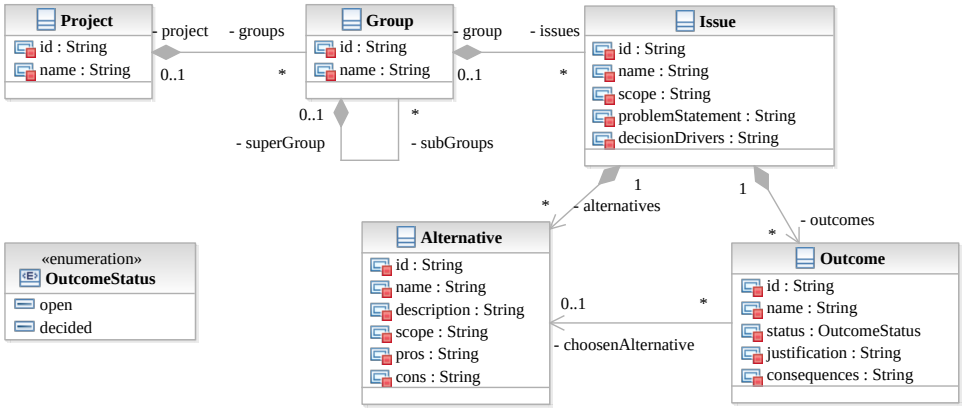

Figure 13: The decision meta model as it is visible at the interface

The domain model for decisions in Fig. 13 is based on the one of AdKWik [ZKL+09], but contains just the most important properties. It may be mapped to other tools as well – the interface is responsible for the mapping of this decision domain model to the domain model of the particular tool. The following is a more detailed description. All design *Issues* of a *Project* might be hierarchically structured in *Groups*. To keep the model simple, only the most important attributes are contained, namely a *name*, the *problemStatement*, and the *decisionDrivers* of an issue. However, we might extend the interface if further information is required. Possible solutions for an issue are given by *Alternatives*, which have a *name*, and advantages (*pros*) and disadvantages (*cons*). *Outcomes* are the result of an made decision; it points to the chosen alternative and has a *name*, a *justification*, *consequences*, and a *status*.

We distinguish the status of an outcome and the status of the binding as shown in Tab. 2. The outcome status distinguished between *open* and *decided*. If an outcome status is decided, the binding status contains the additional information whether the decision is already or is not yet *applied* to the design model. This means in particular that only decided outcomes may be applied. The third column shows an exemplary mapping to the internal status of a concrete decision tool.

Binding status	Outcome status	AdKWik status
open	open	open
decided	decided	decided, confirmed
applied	decided	decided, confirmed

Table 2: Status of bindings and outcomes

Based on this information, the required operations of the interface *IDecisionTool* are listed informally:

- *Get list of projects* available in the decision tool.
- *Get list of groups* for a project or sub-groups from a group.
- *Get list of issues* in a particular group, or by its id,
  (or all related issues, categorized by a dependency type, of a given issue)[11].
- *Create a new group* by providing all required attributes.
- *Create a new issue* by providing all required attributes.
- *Create a new alternative* for a given issue by providing all required attributes.
- *Create a new outcome* for a given issue and alternative by providing all required attributes.
- *Notify about an outcome enforcement.*

The first three functions return data structures as described by the decision meta model in Fig. 13; in order to keep the data transmission concise, projects and groups should not contain sub-elements. All creational functions return the *id* of the newly created element. The notification function does not return anything.

At the moment, *id*s of projects, groups, and issues are assumed to be globally unique; the *id*s of alternatives and outcomes, in contrast, are not required to be globally but only locally unique for the particular issue they are contained in.

5.3 Decision Structure Manager

The decision structure manager handles the coordination between the decision tool and the modeling tool. As already mentioned in the overview, it needs to handle the communication in both directions. It provides four responsibilities to the decision tool (interface *IDecisionStructureManager*):

- *Show the scope of an issue or alternative*, i.e. all model elements that might be affected by that issue.
- *Show the decision structures* for a particular alternative.
- *Show the involved design model elements* of a particular outcome.
- *Apply an outcome* and change the design models accordingly.

All provided functions are asynchronous, i.e. they do not return anything.

These functions provide sufficient information for the decision structure manager to perform its work. For the other direction, the decision structure manager uses the interfaces as defined in Sect. 5.2 and 5.5.

5.4 Decision Structure Library

The decision structure library contains the static structural information for alternatives as well as bindings from outcomes to the respective design model elements. Hence, the provided responsibilities and functions on the interface *IDecisionStructureLibrary* are the following:

- *Store and retrieve decision structures* of alternatives.
- *Store, retrieve, and update bindings* between outcomes and design model elements.

[11] The need for this scenario is matter of further research.

5.5 Modeling Tool

The modeling tool must provide the usual functions for authoring UML models, e.g. creating and maintaining structural and behavioral models and diagrams. In addition, it must provide the following interfaces:

- The interface *IModelAccess* provides access to the models as well as to context-sensitive runtime information, e.g. currently selected model elements.
- The interface *IToolGui* provides the opportunity to extend the GUI with new features, in particular user interactions must be integrated.

5.6 Summary

The rough architecture presented in this section covers the basic components to realize the use cases from Sect. 4. All required interfaces are described informally. Please note that both tools are only extended, i.e. they are loosely coupled and do not depend on each other.

6 Related work

There are several systems and approaches besides the AdKWik [ZGK+07, ZKL+09] which support developers in capturing and making decisions during a software development process.

The ADDSS [CND07] is a web-based tool to collect and store Architectural Knowledge. The primary goal is the documentation and re-use of proven decisions in software engineering processes. In this context, re-use just means looking up already stored decisions and applying them by hand; no automated re-use (e.g. by a tool) is supported. The benefits of this approach, however, are the characterization of decisions and the process integration of decision support.

AREL [TJH07] is another system for the documentation of architectural decisions based on their rationales. It presents a UML profile for modeling architecture rationales and traces them back to the architectural elements. Unlike the work presented here, its intention is not to capture and re-use the changes in the architectural artifacts.

In the Archium framework [JvdVAH07], the system architecture consists of a sequence of design decisions. The archium language is a formal language used to describe these design decisions on a very detailed level which is used for visualization of the decisions and for traceability. But in contrast to our approach, this language does not deal with models but with code.

A similar approach to ours is proposed in [PBR09]; [PBR09] use model transformation specifications to document decisions and build up the system architecture. They leave the specification of transformations open which might be tedious, as well as the process and tool integration of their method.

Technically similar are concepts for design patterns for models which can be found, amongst others, in Borland Together[12] and Rational Software Architect[13]. They also annotate models with pattern-related information and add roles to model elements, provide pattern authoring and pattern matching facilities. However, their patterns are lacking all the different aspects of decisions, e.g. justifications and the separation of problem and solution. Furthermore, the relation between patterns and thus the possibility to propose subsequent patterns are missing.

To conclude, there are many tools (also [BM05, LJA09]) for documenting and reasoning about decisions and capturing knowledge, but proper integration with models of a model-based development process is not covered by any other system so far.

[12] http://www.borland.com/us/products/together
[13] http://www.ibm.com/software/awdtools/architect/swarchitect

7 Conclusion

This report first presented a case study focussing on design decisions in a model-based software developing process. It turned out that decision support for such processes is not yet supported by current approaches. The critical part is the connection between the decision and the design models which is only implicit and *in the designer's mind*. Consequently, the designer has to, on the one hand, consult the decision tool explicitly and, on the other hand, perform the actual changes in the design models manually. A lack of documentation and recurring, manual, and error-prone work are the results.

The use cases set up the requirements for integrating decision management systems with modeling tools. Hence, the user will have immediate access to all design decisions and design knowledge directly within the modeling tool. This allows often recurring decisions and patterns in the models being stored and re-used, which can be seen as best practises. Usability, efficiency, and productivity will be improved, because the design decisions are directly linked to the design models. They give additional information about certain design model elements and even further decisions can be proposed.

Other work focuses on capturing and documenting the design knowledge and rationales for the decisions, but these tools typically are not connected to the models of the process. Thus we try to fill this gap by combining design decisions and design models. The presented conceptual architecture integrates a decision management system with a UML modeling tool by adding an additional decision structure library in-between both tools. We only require the tools to provide the proposed interfaces to work with the decision structure library. One of the important advantages is the tools' independence, since we do not change or extend their meta models. This makes our approach also applicable to other tools than the ones discussed here.

Acknowledgement

The ideas in this paper were compiled with Olaf Zimmermann at a workshop at the IBM Research Lab in Zurich in December 2008 and also revised by Christian Hoertnagl. Many thanks to Ekkart Kindler for lots of helpful discussions and advices.

References

[BM05] Felix Bachmann and Paulo Merson. Experience Using the Web-Based Tool Wiki for Architecture Documentation. Technical Report CMU/SEI-2005-TN-041, Carnegie Mellon University, Software Engineering Institute, September 2005.

[CND07] Rafael Capilla, Francisco Nava, and Juan C. Duenas. Modeling and Documenting the Evolution of Architectural Design Decisions. In *SHARK-ADI '07: Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, page 9, Washington, DC, USA, 2007. IEEE Computer Society.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, Reading, Massachusetts, November 2002.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Massachusetts, 1st edition edition, January 1995.

[IEE00] IEEE Std 1471-2000 IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, 2000.

[JvdVAH07] Anton Jansen, Jan van der Ven, Paris Avgeriou, and Dieter K. Hammer. Tool Support for Architectural Decisions. In *WICSA*, page 4. IEEE Computer Society, 2007.

[Kru95] Phillipe Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50, November 1995.

[LJA09] Peng Liang, Anton Jansen, and Paris Avgeriou. Knowledge Architect: A Tool Suite for Managing Software Architecture Knowledge. Technical Report RUG-SEARCH-09-L01, University of Groningen, February 2009.

[OMG03] Object Management Group. MDA Guide V1.0.1. http://www.omg.org/cgi-bin/doc?omg/03-06-01, June 2003.

[OMG07a] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. http://www.omg.org/cgi-bin/doc?ptc/2007-07-07, July 2007.

[OMG07b] Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*. Object Management Group, November 2007.

[PBR09] Daniel Perovich, María Cecilia Bastarrica, and Cristián Rojas. Model-Driven Approach to Software Architecture Design. In *Fourth Workshop on SHAring and Reusing architectural Knowledge*, pages 1–8, 2009.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In G. Tinhofer, editor, *WG'94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science (LNCS)*, pages 151–163, Heidelberg, 1994. Springer-Verlag.

[TJH07] Antony Tang, Yan Jin, and Jun Han. A rationale-basedarchitecture model for design traceability and reasoning. *Journal of Systems and Software*, 80(6):918–934, 2007.

[ZGK+07] Olaf Zimmermann, Thomas Gschwind, Jochen Malte Küster, Frank Leymann, and Nelly Schuster. Reusable Architectural Decision Models for Enterprise Application Development. In Sven Overhage, Clemens A. Szyperski, Ralf Reussner, and Judith A. Stafford, editors, *QoSA*, volume 4880 of *Lecture Notes in Computer Science*, pages 15–32. Springer, 2007.

[ZKL+09] Olaf Zimmermann, Jana Koehler, Frank Leymann, R. Polley, and Nelly Schuster. Managing Architectural Decision Models with Dependency Relations, Integrity Constraints, and Production Rules. *Journal of Systems & Software*, accepted, 2009.