HOCHSCHULE DER MEDIEN

Diplomarbeit

im Studiengang Medieninformatik

AD_{kwik} – a Collaborative System for Architectural Decision Modeling and Decision Process Support based on Web 2.0 Technologies

vorgelegt von

Nelly Schuster

an der Hochschule der Medien, Stuttgart
am 27.3.2007

1. Prüfer: Prof. Walter Kriha (Hochschule der Medien, Stuttgart)
2. Prüfer: Olaf Zimmermann (IBM Zurich Research Laboratory)

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

(Datum) (Nelly Schuster)

Contents

Abstract

Architectural decisions play a major role in the software engineering process. Each architectural decision and the rationale behind it contain valuable knowledge about the architecture of a software system. Efficient use and reuse of this knowledge is still challenging due to insufficient tool support as well as time and budget constraints. A number of approaches propose ontologies to model decisions in order to efficiently capture architectural decision knowledge. None of these approaches sufficiently covers the architectural decision making process. For instance, they ignore the life cycle of decisions during the decision process, that many relationships and constraints exist between decisions and that the decision making is done in collaboration. This thesis presents AD_{kwik}, a Web 2.0-based system to support the architectural decision making process. To guide through the decision making process, AD_{kwik} leverages a rich user interface designed for collaborative architectural decision knowledge management. Moreover, it facilitates pre-provision of decision content as well as relationship management and decision life cycle based on architectural decision models. Benefits of AD_{kwik} include simplification of decision knowledge use and reuse, acceleration of architecture design project steps, and improvement of decision making and architecture quality.

Keywords: architectural decision, software architecture, architectural decision knowledge management, collaboration, Web 2.0, wiki, Rich Internet Application

1 Introduction

1.1 Context: Architectural Decision Modeling and Making

Traditional documentation of software architecture focuses on capturing information about the design of the solution. Examples are the 4+1 View Model of Architecture [1], the Rational Unified Process (RUP) [2] and the Unified Modeling Language (UML) [3], which are applied to describe the design of an architecture in design models. However, "architectural design, even if well documented [...] is only one small part of the Architectural Knowledge" required for system design, reuse or evolvement [4]. When solely using the above mentioned methods for the documentation of software architecture, important knowledge about how an architecture arose and the rationale behind an architecture get lost.

This knowledge is developed during one important part of (architecture) design: the *decision making process*. The necessity for using a decision-based approach in documentation of software architecture was examined in the 1990s [5, 6], and is again discussed in several recent research publications. Bosch [7] describes software architecture as "fundamentally, a composition of architectural design decisions" and proposes the explicit representation of architectural design decisions in the documentation of software architecture.

There are many definitions of *Architectural Decisions* (ADs). This thesis is based on the definition of Zimmermann et al. [8]: ADs are "[...] conscious design decisions concerning a software system as a whole, or one or more of its core components. These decisions determine the non-functional characteristics and quality factors of the system". They reflect the expert knowledge and rationale behind a certain design. ADs realize one or more requirements of an architecture [9]; they can pertain to one or more elements of an architecture design model [10]. The AD definition does not only cover technological design decisions but also e.g. organizational and strategy decisions. An example AD is 'Platform and Language Preferences', in which the architect has to decide whether to use J(2)EE, .NET or other platforms and languages for the implementation of the system.

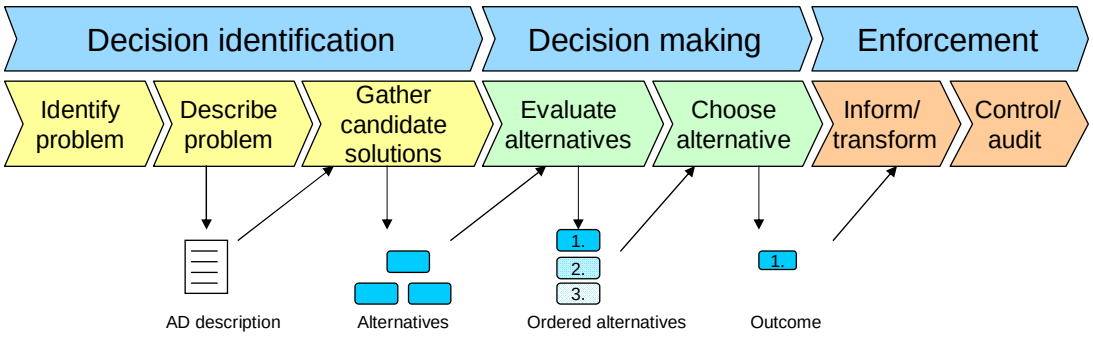


Figure 1.1: The decision making process in software architecture design.

Figure 1.1 depicts the decision making process in architecture design which is adopted in this thesis. The decision making process consists of three phases, namely decision identification, decision making and enforcement [10], which take a sequence of steps. The results yielded in each step are depicted in the bottom row of the figure; they are documented.

Decision identification includes identifying the concrete technical problem to be addressed by the AD as well as recording it. To give an example, the concrete problem is that the design of a software architecture has to be implemented. Thus, the architects have to acquire a suitable platform and language for the implementation. This problem is described and named 'Platform and Language Preferences'. The produced result, the AD description, is shown in the bottom row of the figure. In the next step, candidate solutions for the AD like design patterns [11], technologies, tools or methods are gathered. Candidate solutions for the AD 'Platform and Language Preferences' are for instance 'J(2)EE, Java' or '.NET, C#'. These potential solutions are associated with the AD and called *alternatives*. The alternatives and their pros and cons are documented as well.

The second phase, the *decision making*, is based on results yielded during the first phase. During this phase, several criteria are established to evaluate the alternatives. The criteria depend on the target environment, non-functional requirements, software quality factors and other decisions. Based on these criteria the architects are able to rate the alternatives and finally choose one of the alternatives. The chosen alternative in combination with the reasons why it was chosen is called *outcome*. The reasoning behind this decision, the *rationale*, are expressed through *justifications*.

In the third phase, called *enforcement*, the architect has to ensure, that the chosen alternative is applied correctly on the implementation level during development. This can happen through coaching the developers and reviewing of their code or automatic transformations e.g. from specifications into code. The architect can use the AD and alternative descriptions created during the first two phases of the decision making process to motivate the architecture to the developer.

All three phases of the decision making process are based on personal preferences and experiences, which means in some cases the decisions are more rational, in others less. The decision making can be done unconsciously or after due consideration.

Recording AD descriptions, alternatives and outcomes in a structured form, i.e. as instances of a well-defined model, is referred to as *architectural decision modeling* (AD modeling) [8]. The well-defined model is called *AD model*. Several research publications [4, 12], tools like the Architects' Workbench (AWB) [13] or methodologies like the IBM Global Services Methodology (IGSM) [14] propose ontologies and best practices for modeling ADs and documenting their rationale. They structure an AD model into roughly the following parts:

- AD description – This description contains for instance a problem statement, assumptions about the context and constraints, a recommendation which solution to choose and consequences of the decision.
- Alternatives – Alternatives can be e.g. strategies, patterns, technologies or tools, depending on the type of decision. As alternatives need to be evaluated, most often they are noted with pros and cons.

- Outcome of the AD – This part contains the result of the decision, e.g. the chosen alternative and the rationale for choosing this alternative.

Figure 1.2 shows these three parts of an AD model: There is the AD with its description elements and a number of alternatives from which one is chosen to build the outcome of the AD.

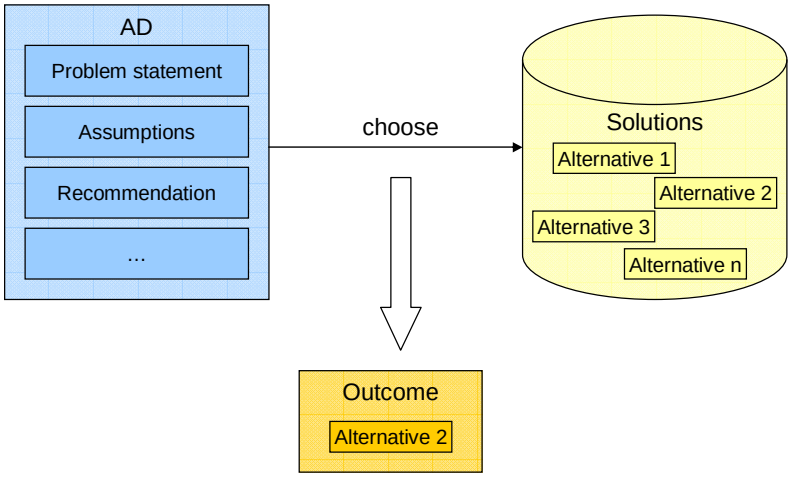


Figure 1.2: Common elements of an AD model.

These approaches expect several advantages of AD modeling which are summarized in the following; the list is not meant to be exhaustive.

- Capture knowledge – An AD can be found behind the usage of every design pattern, technology, etc. The knowledge behind these decisions, e.g. the discarded alternatives, is lost if not captured explicitly [7]. Modeling and capturing this knowledge in ADs and alternatives can help developers while implementing the system as well as new architects while getting familiar with architecture design questions. Maintainers of a system can use documented ADs to understand the decisions and their correlation to own decisions [4].
- Reuse knowledge and experience – Modeled ADs and alternatives can be of help in new projects. Not only can the architects benefit from the experience of other architects, who made their AD models available, but they also can use the AD models as base for their decision making.
- Motivate architecture – Architects who model their ADs are able to promote their architecture to developers through concrete and understandable documents.
- Reveal changes in architecture – ADs relate to parts of the architecture [9]. Dependencies between ADs, and hence dependencies between different parts of the architecture, can be made explicit. Changes in an AD, which result in changes in the architecture, become apparent and help in comprehending necessary changes in architecture. Thus, AD modeling can save money and reduce risk.
- Quality assurance – Modeling ADs observes conceptual integrity of the architecture [9]. The Capability Maturity Model Integration (CMMI) [15] defines best practices for "Decision Analysis and Resolution" [15] (pages 131-144). The proposed process for decision making includes the identification of alternatives for a decision, the evaluation of these alternatives with well-defined criteria and methods as well as the selection of alternatives. If the decision

making process and the modeling of the ADs follows defined methods like CMMI, the architects can proof compliance and quality according to the used method.

However, the decision process includes more than modeling ADs and documenting rationale. Most often more than one person is involved in the process, thus there is also a need for discussion and collaboration.

Based on the definitions summarized in [16] (pages 23ff.) *collaboration* in this thesis is defined as follows: Collaboration is a coordination process with several participants, who want to achieve a common purpose. This process includes discussion and information sharing. Discussion and finding consensus about ADs in a collaboration reveals valuable information: What leads to the AD and its outcome? What are the opinions of different persons about the AD? Who is involved and responsible? Tools can support the collaboration to increase efficiency and effectiveness of persons and groups, especially if participants are distributed over several locations.

Looking at projects in the same domain, e.g. in the development of enterprise applications, one can observe, that the decision process and the relevant ADs in these projects most often are similar. This fact raises the idea of collecting AD models of a domain in a generic but concrete enough manner to provide them to the architects. Pre-provisioning a set of decisions to the architects implies for instance standardization and less need for documentation by the architects. There are several communities of practice (CoPs) [17] in the field of software architecture. In CoPs people collaborate to share experience and expertise, discuss and solve problems in a certain domain. Knowledge of a group is bigger and wider due to different experiences and orientations of the single persons. The knowledge discussed in CoPs can be collected in AD models, as well. For instance for the enterprise application development domain, especially for Service Oriented Architectures (SOA), the collection of AD models with the help of CoPs is already in progress [8].

Figure 1.3 summarizes the aspects of the AD domain explained in this section. AD models are created during the decision making process or pre-positioned, e.g. through the community in a certain domain. The decision making process, which is partly conducted during collaboration and partly by single persons, can make use of the AD models. The decision making process and collaboration can be supported by communities. Communities rely on the expert knowledge gained by persons during collaboration and decision making processes.

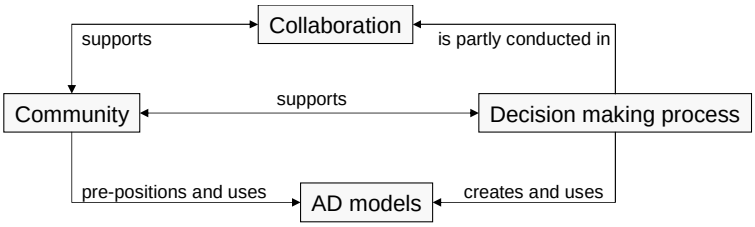


Figure 1.3: Summary of the aspects of architectural decision making.

1.2 Motivation: Tool-support for Collaborative Decision Making

AD modeling is not sufficient to fully support architectural decision making processes: Besides the need for pre-provisioned AD models, collaboration has to be facilitated and documented as well.

Furthermore, a powerful mechanism for collaborative AD knowledge management is required which makes AD knowledge explicit. Surveys like the Survey of Architectural Design Rationale [18] confirm that the awareness to document and reuse ADs and the rationale behind the decisions is given. But the survey also shows, that the intention to document is weak. Table 1.1 makes clear that one third of the architects do not capture ADs because they lack a suitable tool. The major part of the architects state that time and budget reasons detain them from documenting ADs. Another important reason are the missing standards for architectural decision modeling.

Topic of questions	Percent of respondents	Number of respondents
No standards	42%	34
Not aware of	4.9%	4
Not useful	9.9%	8
No time/budget	60.5%	49
No suitable tool	29.6%	24

Table 1.1: Reasons of not documenting design rationale [18].

This thesis aims to alleviate these problems by making ADs first class entities in the software architecture design process. It presents a tool-supported concept with prototype implementation to assist the architects with an AD knowledge management system to capture, organize and reuse AD models. Pre-positioned AD models, as well as collaboration mechanisms are also part of the concept and facilitate the architects in the decision making process. As consequence, the three main reasons of not documenting ADs are tackled: The tool support focuses on suitability and usability to the domain of the architects. Pre-positioned AD models as well as support of team collaboration and the decision making process accelerate architecture design project steps, thus allow the architects to save time and budget. Standardized and structured AD models as well as the pre-position of AD models improve AD making and result in architectures of higher quality.

1.3 Structure of the Thesis

The remainder of this thesis is structured as follows. After this first introductory chapter, the second chapter introduces technology concepts which are essential for understanding this thesis. Chapter 3 includes the analysis of the functional and non-functional requirements as well as an analysis of candidate frameworks for the implementation of the concept. The requirements are gathered using a simple scenario. Chapter 4 introduces the proposed solution which is driven by the requirements of the previous chapter. Chapter 5 outlines details of the implementation of the prototype. Chapter 6 continues with an evaluation of the solution using the requirements, existing work and first user test results. Finally, in Chapter 7, the thesis concludes with a retrospective look at the development, a glance into the future work and a summary.

2 Base Technologies and Principles

This chapter gives a short introduction to concepts and technologies used throughout the thesis. It includes an explanation about architecture layering, client paradigms, namely the rich and thin client paradigms, collaboration and collaboration tools as well as Web 2.0 technologies like wiki and Ajax. These introductions aim to solely provide knowledge needed in this thesis. Thus, they are not meant to be exhaustive.

2.1 Architecture Layering

Layering of application architectures is a technique to "break apart a complicated software system" [19]. The layering technique can be applied in different levels of abstraction: logically and physically. Logical layers are used to decouple responsibilities, which allows the usage of one layer without having to fully understand the implementation of other layers. As each layer can be seen as a whole, the replacement of layers with other implementations becomes easier. Dependencies between layers are minimized. This facilitates reusability, maintainability and scalability. On the other hand, strict layering can reduce performance, since more components are involved and more interactions become necessary between layers.

The three logical layers used throughout this thesis are presentation, domain and data source [19].

- **Presentation** – The responsibility of the presentation logic is the management of interactions between the user and the system. The presentation logic displays information and provides services for the user, e.g. dialog and page flows in Web applications.
- **Domain** – The domain layer is responsible for computing logic in the domain, e.g. business logic computations, integrity checks, workflow logic or credit checks in financial applications.
- **Data source** – The data source layer is responsible for the storage of persistent data and the access to the storage.

Logical layers can be distributed to different physical nodes, which are also called *tiers*. Tiering, i.e. physical layering, allows the specialization and thus optimization of computers for one task. This thesis adopts the idea of a 3-tiered architecture for design and implementation. Following the naming conventions of [20], the three tiers are client tier, application tier (or mid-tier) and data tier. The client tier runs on the user machine and interacts with the user. The application tier contains the application logic. The data tier includes the data, e.g. in a database. A tier itself can consist of several computers, but also can several tiers reside on the same machine. Application and data tier commonly are summed up under the term server, as they provide services for the client tier.

According to Fowler [19] mainly two paradigms are in use for the distribution of the logical layers on the client and application tier: the thin and the rich client paradigm. The next section introduces and compares these two paradigms.

2.2 Rich Client vs. Thin Client

In a thin client application, all three logical layers are located on the application tier. The client does not contain any presentation logic; it is only responsible for presenting views on the data and delegating user requests to the server. An example application is a Web application which is written in plain HTML, e.g. simple email service. The architecture of a Web application is depicted in Figure 2.1. The client, i.e. the Web browser, follows a page flow. Whenever the user requests a new view of the data, e.g. selects an email in the overview of all emails on the Web page, the Web browser delegates the request to the Web server. The result of the computations on the server, e.g. an HTML page which contains the email details, is sent back to the client. The client simply presents the retrieved page to the user. This means, whenever the user requests information, the server needs to be consulted. This fact reveals a restriction of the interactivity of applications realized with the thin client paradigm. The administration effort of thin clients usually is lower than the administration of rich clients, as all computations happen at one place, the server. Another advantage is that the installation is easy, as local resources on the users machine are utilized, e.g. the Web browser.

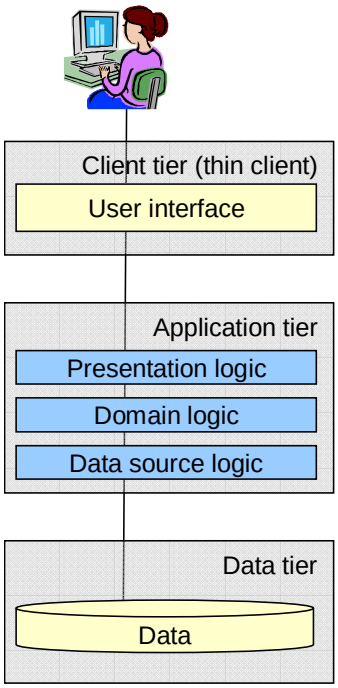


Figure 2.1: Thin client paradigm: All logical layers are located on the application tier.

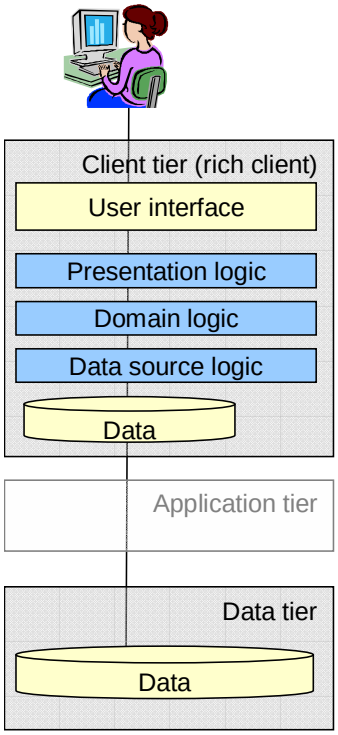


Figure 2.2: Rich client paradigm: The client tier contains logical processing.

In a rich client, the client tier is used to process parts or all of the logical computations. In some client/server architectures, even the application tier is not needed, since all computation is done on the client. In others, at least the presentation logic is processed on the client. An example rich

client is an email client like Microsoft Outlook or Thunderbird. As computations are done on the rich client, a richer user experience is possible, i.e. the user's perception of the ease of use and overall quality is better and the users perceive the interaction design as more powerful. Examples are field validations in input forms, e.g. spelling checks for newly created emails, or disabling requests that are not reasonable in the current context. A request does not necessarily result in a reload of the whole application. For instance, if the user selects an email in the list of all emails, the email rich client only needs to reload the part of the application which displays the email details. Figure 2.2 depicts a rich client, which processes all logic. The user of such rich client applications can work with the application data without permanent connection to the server. If the user selects an email in the email rich client for presentation, the client accesses the local copy of the email. If the domain logic is located on the client, updating and maintaining an application is getting more difficult since every client needs to be updated.

An email application – whether accessed with a thin or rich client – is a popular collaboration tool used in e-collaborations. A definition of e-collaboration as well as knowledge management is given in the next section. Moreover, the section introduces further collaboration tools which support the e-collaboration.

2.3 E-Collaboration and Collaboration Tools

E-collaboration and collaboration tools are discussed and developed in the (research) area of computer supported cooperative work (CSCW). This area aims to understand and support the cooperative, collaborative and communicative work of two to many persons. According to Wilson [21], CSCW focuses on two main components: underlying technology and group process issues, i.e. social aspects. Applications in the field of CSCW cover conference systems like electronic meeting rooms, decision support systems, multi-user editors like whiteboards and messaging systems like email or chat. This is only a short list of applications; the field of CSCW is a wide area.

Collaboration is defined earlier in this thesis as a coordination process with several participants, who want to achieve a common purpose. E-collaboration is defined in this thesis as Web-based collaboration. Web-based collaborative tools are used to support e-collaboration.

According to Haas [16] (page 248), there are two main tasks of collaboration tools: Ease of communication and improvement of knowledge management. Knowledge is a "fluid mix of framed experience, values, contextual information, and expert insight that provides a framework for evaluating and incorporating new experiences and information. In organizations, it often becomes embedded not only in documents or repositories but also in organizational routines, processes, practices, and norms" [22]. Knowledge management uses practices and technologies to capture, organize, reuse and make this knowledge explicit.

As the target audience of this thesis are persons involved in the architecture design and software development process, the following list describes tools used in the e-collaboration of architects and software developers.

- Email: Via emails any sort of information can be exchanged or discussed. This can happen almost in real-time, if wanted. Thus, communication is facilitated. The organization of content is left to the user. A number of email software supports deleting useless emails and sorting single emails into different folders. However, text contained in emails usually is not structured. As it is a popular communication medium, users receive and send many emails concerning various topics. This makes extracting of knowledge and knowledge management difficult.
- Instant messaging/chat room: Through instant messaging, groups can discuss in real time. This enables effective communication and decision making. Knowledge is captured in message logs without management. Extracting knowledge of message logs is left to the user.
- Weblog/Blog: A Weblog is a Web page in diary style, which includes information about a certain subject. Often it represents the opinions and experiences of one person, the author. A Weblog is updated periodically. Information in Weblogs is communicated in mostly one direction, from the author to the audience. Knowledge management is conducted by the author.
- Message board/Internet forum: The focus of message boards is on discussion as well as asking and answering questions about a particular domain or topic. This enables smooth communication. Knowledge management is improved through search functionality or the organization of messages into topics.
- Teleconferencing: A teleconferencing software supports ad hoc conferencing with voice, chat and whiteboards. Communication like in real world is possible. Drawings from the whiteboards can be organized. As discussed above, the management of knowledge exchanged in chat discussions is complicated. Knowledge included in voice is lost, if not recorded.
- Wikis (see Section 2.4): Wikis provide the possibility to share community knowledge. Communication depends on the type of the wiki, as wikis also can include message boards or other collaboration features. Knowledge typically is organized by the community.
- Portals: Portals collect resources and services mostly for a special domain. They can provide several of the above mentioned applications. Communication and knowledge management possibilities depend on how the resources are structured and which services are included.

Two of these collaboration tools, Weblog and wiki, are so called Web 2.0 applications. The next section elaborates on the Web 2.0 paradigm. Moreover, it explains wikis in more detail and introduces situational applications which can be used as platform to build customized collaboration tools.

2.4 Web 2.0, Wiki and Situational Applications

Web 2.0

The definition of the term Web 2.0 has being intensively discussed until and after O'Reilly released his definition of Web 2.0 [23]. This definition contains several aspects which are important for this

thesis. First of all, Web 2.0 is understood as an “attitude, not a technology”. It provides a platform for the development of applications, which are improved continuously with the contribution of the users. The more users an application has, the better it gets. This leads to another important aspect of Web 2.0: The “harnessing of collective intelligence”, i.e. the usage of the aggregated knowledge of the user community. Active involvement of the users who add content and connect it through hyperlinks, rate products or discuss about different topics lets the Web become a “global brain”. Web 2.0 builds an “architecture of participation” [23], e.g. through communities of practice.

Besides the users, data plays an important role in Web 2.0. Data is more important than the applications themselves. It has to be collected, interconnected and enriched with meta data. Then it can be provided via an application programming interface (API) or through front-ends which provide rich user experience similar to desktop applications. These front-ends are called Rich Internet Applications or RIAs. They are similar to thin client applications as they usually run in Web browsers, but they can process data on the client an thus provide the user with more interaction possibilities. RIAs can be realized with technologies like Ajax [24], which is shortly explained in Section 2.5, or Adobe Flash [25].

Two application concepts of Web 2.0 which are used in thesis are explained in the following: wikis and situational applications.

Wiki

A wiki is a collaboration application which basically forms a knowledge management system. It is a collection of readable and writable Web pages which contain cross-links to other pages [26]. The main idea of wikis is the sharing of knowledge, information, ideas and experiences. Wikis follow the paradigm of open editing which means, everyone can add, edit, and delete content. The content belongs to the whole community of the wiki, not to the person who created it. To ease the open editing, most wikis are Web browser based. This allows for easy access for the users. Wikis follow the idea of simplicity. Contribution should be as simple as possible, which ensures good usability and acceptance. Another important paradigm is trust of the users in other users. The maintenance of content by the users ensures that most users act to the good of the community. Therefore many wikis do not implement quality assurance mechanisms. Some wikis provide user management, page or author ranking or vandal patrol, which for instance could detect advertisements.

The first wiki, WikiWikiWeb [27], was created by Cunningham to support software specialists during the development process. It focuses on patterns and processes used in software development. Today’s most popular wiki is Wikipedia, a free encyclopedia which contains several million articles in more than 100 languages [28].

So called enterprise wikis are used in enterprises to manage and share knowledge. These wikis typically provide sophisticated user access control. Some enterprise wiki software also provide APIs to integrate the wiki as knowledge management system into other enterprise products.

There are several mechanisms which extend the conventional wiki. Structured wikis like TWiki [29] provide the possibility to structure content on the wiki pages similar to a database. Others

allow structuring pages in hierarchies. In semantic wikis like Makna [30] the users can enrich the content with meta data. This opens new possibilities in search functionality and organization of the content. Application wikis like JotSpot [31] can be mashed up with *widgets*. Widgets in application wikis are small programs which can be placed on a wiki page to provide functionality like calendars, address lists or maps. This can lead to so called *situational applications*.

Situational Applications in the Web (Mash Ups)

A situational application is a software which is created by the users. It typically is very specific for the domain and time in which it is developed. Situational applications in the Web are also called *mash ups* which are Web sites or applications combining content of different sources. Application wikis for instance provide widgets which the user simply can include into wiki pages. The widgets can be configured in a way that they collect data from different sources. They format the data and display it on the page. Users easily can combine these widgets to build collaboration platforms fitting their needs.

As mentioned, for the implementation of such Rich Internet Applications, technologies like Ajax are applied. The next section gives a short introduction to Ajax and the difference between conventional and Ajax-enabled Web applications.

2.5 Asynchronous JavaScript and XML (Ajax)

Conventional Web applications (see thin clients in Section 2.2) use a synchronous request-response mechanism for interactions between client and server. Whenever the user of the Web application presses a button on the Web page, the client sends a HTTP [32] request to the server. The server processes the request and returns a new Web page to the client. While the server processes the request, the client, thus the user of the Web application, has to wait for the response. During this time, the user can not issue a new request to the server. The left side of Figure 2.3 shows this interaction pattern.

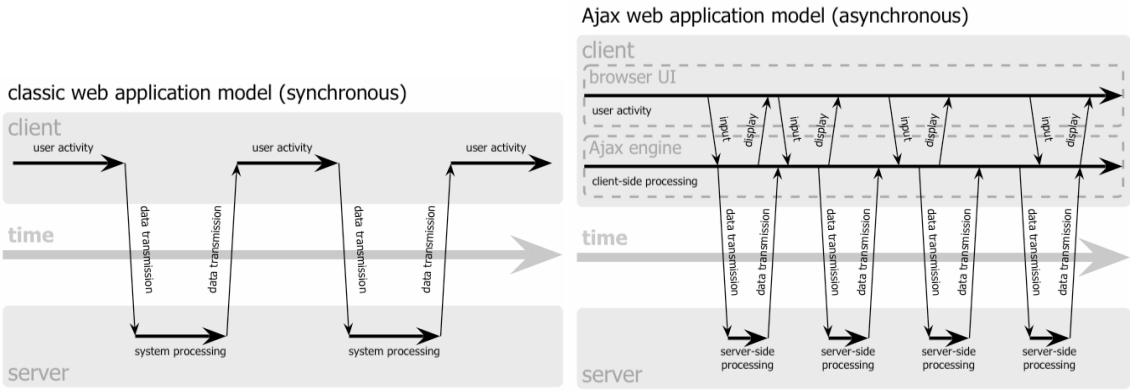


Figure 2.3: The synchronous interaction pattern of a traditional Web application (left) compared with the asynchronous pattern of an Ajax application (right) [24].

Ajax (short for asynchronous JavaScript and XML) is a combination of several Web technologies to develop interactive Web applications like RIAs. Ajax can be used to send client requests to

the server without reloading the whole Web browser page. This is performed using an additional processing layer which is included into the Web browser, the Ajax engine. This engine, which is written in JavaScript, can directly communicate with the server and, at the same time, render parts of the user interface. The Ajax engine enables asynchronous communication between client and server. This means, the user can interact with the client without having to wait for responses of the server. The interaction pattern of Ajax applications is shown on the right side of Figure 2.3.

Response time, i.e. the duration between sending a request and receiving the result, is important for the usability of an application, whether desktop or Web application. If an application needs more than ten seconds to respond, the user wants to change to another activity during waiting time [33]. The asynchronous communication in Ajax Web applications enables the reload of single parts of a Web page. This results in faster response times and thus in better usability, as not always the whole page is reloaded like in conventional Web applications. Furthermore, the Web page is not blocked while computing the response. As the presentation logic partly resides on the client, the page flow as found in conventional Web applications can be enriched with better interactivity.

The previous sections give a short introduction into paradigms and technologies used throughout the thesis. The next chapter analyzes the requirements for the design of the system presented in this thesis. It then evaluates candidate assets for the implementation of the prototype.

3 Requirements and Candidate Asset Analysis

Chapter 1 motivates a concept for collaborative AD knowledge management and outlines the need a usable tool to support decision processes. The functional and non-functional requirements for such system are examined in the first section of this chapter. The technologies and principles explained in Chapter 2 are particularly important for the understanding of the second section of this chapter: This section evaluates candidate assets one of which will support the implementation of the prototype.

3.1 Requirements Analysis

After a short analysis of the target audience in Subsection 3.1.1, Subsection 3.1.2 introduces the scenario which is used to capture the requirements. Subsection 3.1.3 outlines use cases in the scenario. The use cases lead to technical requirements in Subsection 3.1.4. Subsection 3.1.5 outlines the non-functional requirements.

3.1.1 Target Audience: Software Architect

This thesis uses the role of the software architect as target audience. However, the concept is not restricted to software architects but can also be used by other types of architects. Its usefulness depends on the reusability factor of ADs in the domain of the particular architect. Assuming that an architect – independent of the domain – has to reflect about the same ADs in every project he performs, the application of the concept will support him.

A software architect is responsible for turning the evaluated requirements of the system under development into an architecture. On the one hand, a software architect needs the ability to keep an eye on the big picture. On the other hand he needs to be proficient in actual technologies and methods. One main responsibility of the architect is to ensure the smooth interaction of technologies, which can also mean implementing prototypes, testing tools or reviewing patterns and keeping track of the newest technologies. Another key responsibility of a software architect is considered decision making under time pressure. This all makes the decision making process depicted in Figure 1.1 on page 6 an integral part of the software architect's activities.

Besides deep knowledge in technology, the role of the software architect requires capabilities like sound communication skills and organizing ability. The software architect needs to motivate his architectural solution and to guide the developers into the right direction. In a team of architects, a lead architect also has to coach, which requires good leadership competencies. Furthermore,

the software architect needs to understand the business strategy and the organizational politics in order to create solutions which are useful [34] from a business perspective. All these skills require experience – there is no overnight software architect.

Software architects most often are members of a team of developers, product specialists and other architects. Nowadays teams can be spread all over the world but still need to collaborate. Collaboration often happens via phone or Web-based collaboration tools which are shortly explained in Section 2.3.

Figure 3.1 summarizes the aspects which are covered by the role of the architect.

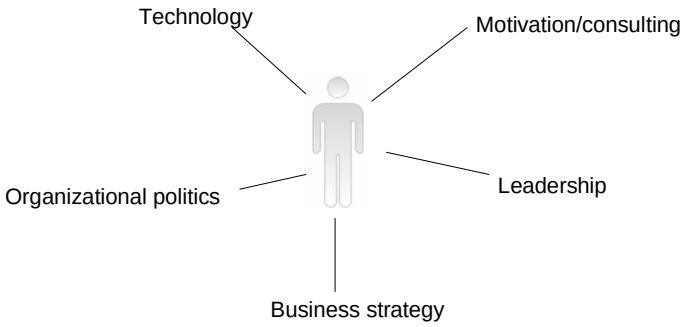


Figure 3.1: Activities and competencies of an architect (cp. [34]).

Due to the training required for certain certifications for the architect profession (e.g. IBM Global Services Method (IGSM) [35] (pages 18ff.)), the way how architects work and the terminology can be standardized. IBM architects for instance, who use the IGSM, should be used to document ADs as argued in the ARC 100 work product [14]: “There are no reasons for not documenting system architecture and design decisions”. However, as Table 1.1 shows, this is not consistent with reality.

To illustrate the requirements for the desired system to support the architects in architectural decision making, this chapter makes use of a scenario in the SOA domain. The following section introduces background information about the example.

3.1.2 Scenario Introduction

Company XYZ is a software development company employing more than 1000 people who are spread over several locations all over the world. XYZ’s focus is on custom application development and integration.

Several software developers and architects of XYZ express the need for a better support of their software development process. The developers complain about the incomplete description of the architecture they should implement. The architects mention difficulties in documenting their architecture in a way that the information about the architecture is reusable in other projects. Enterprise lead architect Lucy expresses the wish to effectively communicate her enterprise-wide architecture decisions to the project architects.

A team of several architects in XYZ therefore is instructed to analyze the situation and to come up with a solution. Indeed, the project team under the lead of Peter identifies the need for a concept to support the decision making processes of the company's architects. To analyze the requirements for the concept, the project team members examine their own method of working which is typical for the design and development of software solutions in XYZ:

Typically, several architects are involved in designing the system. As mentioned, they are spread over several locations complicating their collaboration. Lucy's enterprise-wide decisions are already documented and stored in a tool. Lucy is not involved in the software development projects directly. Nevertheless, her decisions ought to be obeyed in all projects. The decision support approach should be integrated into the established software development process and used tools.

In order to enable Web-based collaboration and decision making of the architects, Peter and his team decide to develop a front-end, which manages their decisions regarding the architecture of a system. Considering the requirements and their solid experience in the principles of SOA, Peter and his team will adopt these principles for the development of the decision front-end system.

The next sections outline the requirements of the system by analyzing the use cases and showing decisions which Peter and his team come across while designing the decision front-end.

3.1.3 Usage Scenario as Use Case Model

The typical development process of a project in company XYZ includes three phases: the *project initiation*, the *solution outline and design* and the *project closure phase*. This subsection outlines the functional requirements captured as use cases in these phases. It explains how the architects could use the tool-supported concept during the development of a system, e.g. the decision front-end.

Use Cases in the Project Initiation Phase

Figure 3.2 shows the use cases in the project initiation phase. Three use cases cover the user management in the application: log in to and log out from the system and manage users, e.g. administrate user rights or create new users. These use cases are present during all phases of a project. Use case inspect AD content is also part of all three phases and applied by all architects. The following explains use cases which are specific to the project initiation phase.

UC100 Import ADs from other sources.

To pre-provide AD content in the front-end, the application needs a possibility to import ADs from other sources into the application. This is also called *pre-population*. In the scenario Peter wants to use the existing decisions of Lucy, which provide a basis for the decision process of the system development team. Lucy uses the Architects' Workbench (AWB) [13], a tool for managing information associated with the design and delivery of IT architecture. Thus, the import of AWB sources into the application is one example for this use case.

UC200 Adopt the AD content.

After importing the ADs into the application, Peter has to sort out all ADs which are not relevant to the decision front-end development. This can be summarized under the term adopt the AD content. That means, Peter needs to browse through the imported ADs, look at the descriptions and delete the ADs not needed. For instance, in the requirements analysis the team evaluated that no business processes need to be implemented. Peter therefore removes all ADs which are related to business processes. This includes e.g. 'BPEL Engine', the decision about the runtime container for processes.

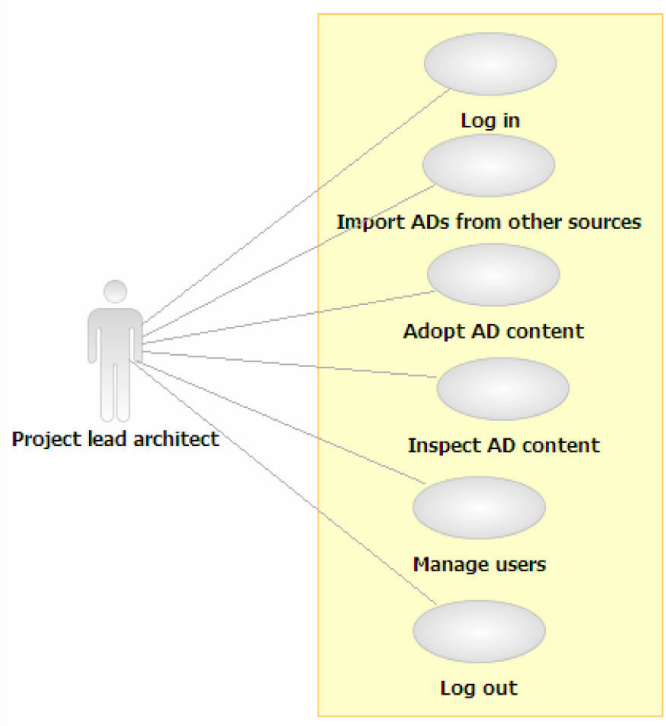


Figure 3.2: Main use cases in the project initiation phase.

Use Cases in the Solution Outline and Design Phase

After release of the ADs to the other architects of the team, all architects and product specialists can start working with the adopted AD content. This denotes the solution outline and design phase. The use cases in this phase are depicted in Figure 3.3.

UC300 Inspect AD content.

Based on the structure of and the dependencies between the ADs, the architects want to navigate between different AD descriptions and display them. This use case includes displaying the alternatives of an AD or dependent ADs.

UC400 Document AD.

During the development process, the architects gain more knowledge about domain and existing technologies. They will use the existing ADs to guide them, but they will also find ADs which

are not available in the pre-populated front-end. The architects therefore need to document their decisions during the project. The documentation of ADs includes several use cases which are explained in the following.

UC410 Create AD.

The first step of documentation is the creation of a new AD. For example, one architect identifies the decision about the presentation layer technology, which is not in the front-end yet. He therefore creates a new AD called 'Presentation Layer Technology' and adds the alternatives 'Rich Client' and 'Thin Client' to this AD.

UC420 Edit AD.

As the knowledge of architecture and ADs is developed continuously, AD descriptions need to be modifiable. It might happen that the architect identifies a new alternative for an AD, e.g. in discussions with other architects or developers. He then needs to change the description of this AD. Adding alternatives to and dependencies between ADs are special cases of editing an AD. In the example, several architects mentioned during discussions, that RIAs are also a possible solution for the presentation layer. One architect therefore adds the alternative 'Rich Internet Application/Web 2.0' to the AD 'Presentation Layer Technology'.

UC430 Move AD in content.

Some of the architects like to create all ADs they have in their mind at once, and then organize them into the existing AD content. In the scenario, Lucy ordered her decisions into different topics. One topic is 'Application Front-End Decisions', which contains all decisions related to user interaction. The newly created AD 'Presentation Layer Technology' is not yet ordered into that topic, although it logically belongs to it. One architect therefore moves it to this topic.

UC440 Upload documents.

To supply further information about one decision the architects want to contribute documents like whitepapers, diagrams and pictures and associate them with an AD. Therefore there must be a possibility to upload documents and assign them to one AD.

UC450 Delete AD.

If an architect created an AD which seems to be unnecessary or even false, there must be a possibility to delete it.

UC500 Document AD outcome.

If a decision is made, the architects want to document the outcome including the chosen alternative, a justification and information of date and responsible person. The outcome description shall be available for displaying. If it is not clear why a certain alternative was chosen, the architect simply can lookup the justification of the outcome. In the requirements analysis for the decision front-end, Peter and his team captured the alternatives 'Rich Client', 'Thin Client' and 'Rich Internet

Application/Web 2.0' of the 'Presentation Layer Technology' decision. They decide for 'Thin Client', since thin client applications are easier to install and maintain. Therefore, they document this alternative and justification as outcome for the AD.

UC510 Revise AD outcome.

If an outcome turns out to be wrong or based on unrealistic information, the decision has to be reconsidered. In particular cases a decision outcome even might be revised. In the example, the architect who is responsible for the user interface recognizes, that a thin client does not provide a powerful enough user interface. Providing this reason, he therefore rejects the outcome of 'Presentation Layer Technology', which was decided in favor of 'Thin Client'. He then documents the new outcome 'Rich Internet Application/Web 2.0' with the justification 'provides a richer user interface than thin client, but maintenance and installation can be compared to a thin client application'.

UC520 Approve AD outcome.

An outcome has to be approved by the project lead architect. This can lead to rejection of a decision by the lead architect. In the example, Peter is satisfied with the decision for 'Rich Internet Application/Web 2.0', since the justification is coherent. He therefore approves it.

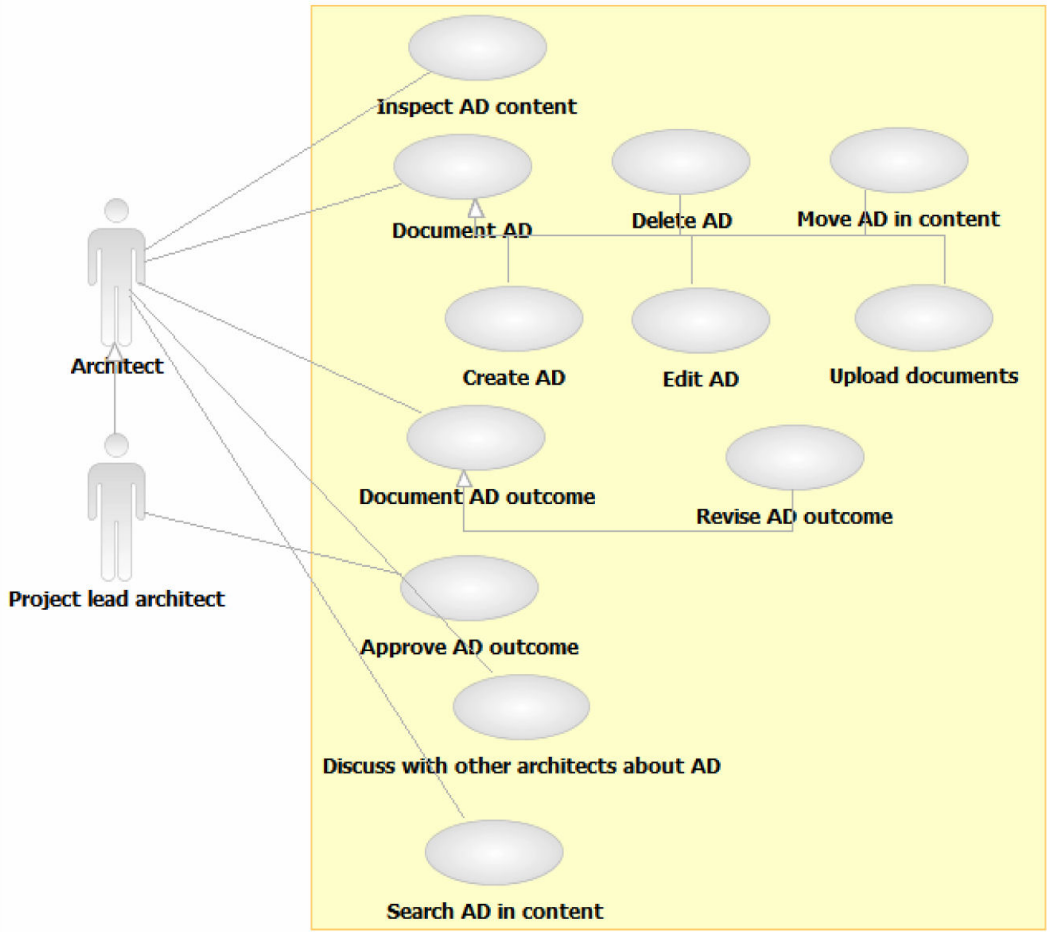


Figure 3.3: Main use cases in the solution outline and design phase.

UC600 Discuss with other architects about AD.

Identifying and documenting ADs and making decisions happens collaboratively, which means architects want to discuss and comment ADs which are captured in the front-end. To enable this kind of communication, a discussion feature is needed. In the scenario, the discussion about alternatives for 'Presentation Layer Technology' is done via the front-end.

UC700 Search AD in content.

The front-end must provide a possibility to find identified and described ADs. The architect who is responsible for ADs about the user interface searches for the term 'user interface'. He will then receive all ADs related to this term, e.g. the 'Presentation Layer Technology'.

Use Cases in the Project Closure Phase

In the project closure phase, the team decides to provide its gained AD content to follow-up projects as well as the enterprise architect for reviewing. Figure 3.4 shows the use cases in this phase.

UC800 Anonymize AD content.

As parts of the content might be project specific, containing confidential information of the customer or some decisions which are very specific, the team wants to clear the content before providing it to other teams. A very important feature is to replace terms, e.g. the company name of the customer, with a pseudonym in all ADs, alternatives, outcome descriptions and comments. It must also be possible to delete ADs which are too specific or confidential.

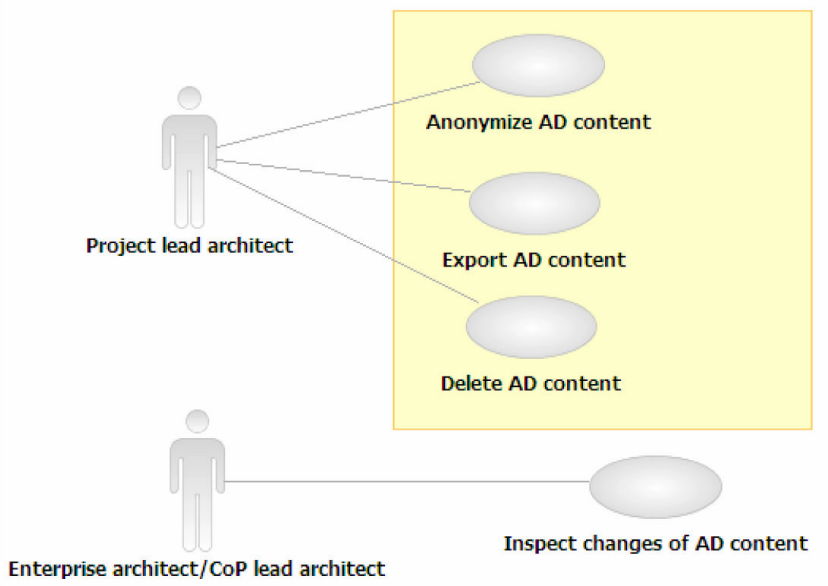


Figure 3.4: Main use cases in the project closure phase.

UC900 Export AD content.

After successful design of the solution, Peter wants to deliver the decision content to the enterprise architect Lucy who can revise the enterprise wide decisions with the help of the new decisions. The content therefore has to be converted back into the format of Lucy's tool. Lucy, representative for enterprise or CoP lead architects, uses her own tool, e.g. AWB, to compare her decisions with the decisions made in Peter's project. Figure 3.4 shows this in use case inspect changes of AD content.

UC1000 Delete AD content.

After successful closure of the project the system can be cleaned up by deleting the whole decision content from the application.

Summary of Use Cases

Figure 3.5 summarizes the use cases and their context. The arrows denote AD knowledge flow. In the project initiation phase, the project lead architect adopts the AD knowledge he gets from the enterprise architect or CoP lead architect. In the second phase, the solution outline and design, he delivers the adopted AD knowledge to his team. In this phase, the AD knowledge is discussed and documented. It then is handed over to the last phase in the project, the project closure phase. In this phase, the AD knowledge is cleaned up. Then, the circle is closed by exporting the AD knowledge back to the enterprise architect or CoP lead architect.

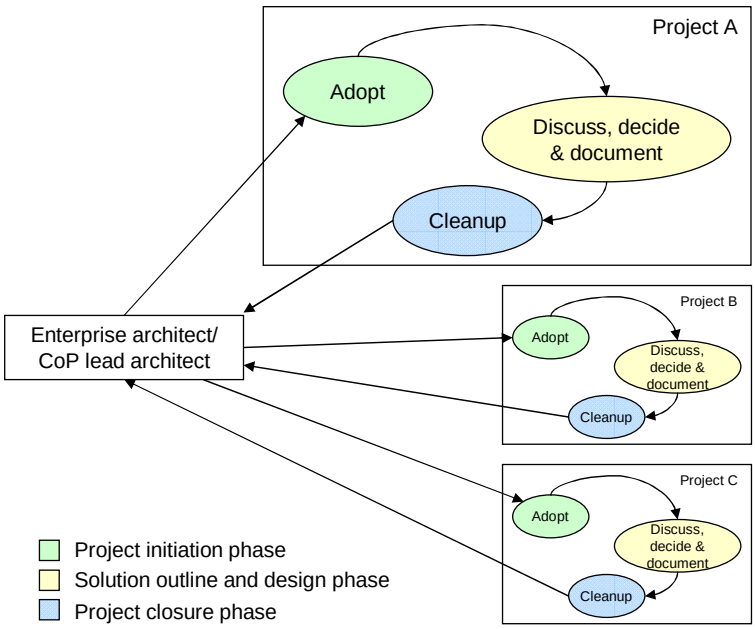


Figure 3.5: Summary of use cases and context: asset harvesting.

As depicted in the figure, in this way every project contributes to the collaborative AD knowledge. This collaborative AD knowledge management effort through projects is also called *asset harvesting* from projects.

The implementation of the use cases in a tool requires several technical function blocks. These technical requirements are explained in the following.

3.1.4 Technical Requirements

This subsection presents the technical requirements (TR) which result from the analysis of the target audience (Subsection 3.1.1) and the usage scenario (Subsection 3.1.3).

TR100 Domain Model for Architectural Decisions

In order to model ADs a domain model [19] is needed which provides a well-defined structure for ADs, alternatives and outcomes. In fact it is more convenient and efficient to read and understand well-structured than plain text documents. Structured content provides the ability to extract the needed information easier, which is important for readers under time pressure. It also ensures an increased reusability, because more architects are willing to read through structured descriptions. Modeling the AD domain with a domain model enables automation, since all instances have the same structure.

The following example shows how such a structure could look like. It describes one of the ADs which Peter and his team needed to decide in the scenario: the ‘Presentation Layer Technology’.

> ‘Topic: Presentation Layer Technology
>
> Problem Statement: Users of an application need to be able to assess and manage the application through a user interface. Without a suitable presentation layer technology, applications will not be able to deliver much business value.
>
> Alternatives:
>
> 1. Thin Client
> 2. Rich Client
> 3. Rich Internet Application / Web 2.0’

This description uses two parts of the domain model, the AD description and the alternatives. The AD description shall contain meaningful elements which describe the AD. The challenge when designing the AD description is to find the right granularity. Insufficient structure counters reusability, while too much restriction can lead to unusable descriptions or lost acceptance. Besides the AD description, the domain model shall provide a structure for alternatives, outcomes, dependencies and the organization of ADs.

TR200 Dependency Management

Considering the scenario and the previously described example AD it becomes clear that the application also has to handle *dependencies* between ADs. This means if an architect chooses a ‘Thin Client’, he does not need to decide about how to support an installation routine. If the architect

decides for a 'Rich Client', one of the next decisions will be how to provide the installation routine: e.g. provide it as download from a Web site, save it on a disc or use the update mechanism of the used framework, e.g. the Eclipse Rich Client Platform. The system therefore has to implement a powerful concept for *dependency management*.

Dependency management can be used to hide ADs if they become obsolete; e.g. if the architect chooses 'Thin Client', the AD 'Installation Routine' will disappear. Another purpose is the guidance of architects through the decision process. Based on the dependency management, the tool can lead the architect: Which ADs need to be decided in the next step? Which ADs are obsolete? It can also help the architects to do an impact analysis of decisions, e.g. through simulation of special outcomes.

The visualization of dependencies on front-end basis is required. The concept must be generic enough to allow future work to extend the dependency management to other usage scenarios.

TR300 Content Repository

Assuming several architects have collected ADs in former projects which should be reused in current projects, a knowledge management system is required where they can store, edit and share their documents. This requirement addresses UC400-UC500, the documentation of ADs, alternatives and outcomes. The repository shall store ADs, alternatives and outcomes which are all instances of the required domain model.

A content repository represents an information management system which provides services transcending the storage of data in traditional data repositories like databases or file systems. The following list includes the services which are needed by the target audience, the architects.

- **TR310 Data Storage** – The documents must be stored in a memory, which supports AD documentation. Documentation consists of well-structured documents describing decision content, i.e. instances of the domain model, and several additional assets like research papers, technical reports or diagrams.
- **TR320 Versioning** – The repository is used collaboratively. To keep track of changes and provide the ability to roll back changes in a description, the content repository must provide a versioning mechanism.
- **TR330 API** – An application programming interface (API) provides the ability to access functionality of the application through source code. The content repository must provide an API to access, create and edit the data from outside the application.
- **TR340 Import/export mechanism** – UC100 and UC900 require an import and export mechanism to and from the content repository to avoid manual transfer of the documentation between two applications.
- **TR350 Search** – To support the anonymization of decision content (UC800), the content repository must provide a search and replace functionality. Furthermore, UC700 makes clear that a conventional search functionality is needed, as well.

- **TR360 Document generation** – To allow for reuse of all descriptions, e.g. as reference for developers or presentation manuals for customers, the generation of text documents from the AD, alternatives and outcomes is required. This should include the generation of IGSM ARC 100 [14] work products in HTML or Word format.

TR400 Decision Workflow

UC500 advises the need for the support of the decision process. Besides the possibility to document outcomes, the decision workflow shall support the steps of the decision making process which are depicted in Figure 1.1 on page 6: The identification of the problem, the collection of possible alternatives, the selection of one alternative as outcome and the enforcement of the outcome.

TR500 Collaboration Features

As mentioned in the scenario description, the architects collaborate from different locations. Peter is located in the US while the others are in Switzerland. The only medium they can collaborate efficiently is over the Web.

To ensure that more than one person can use the application at one time, it must support multi-user access. It has to provide an authorization and log-in mechanism to track the activities of the individual architects. Furthermore, the tool shall support collaborative work, i.e. provide possibility to communicate over the front-end in a useful manner like described in UC600. If information is exchanged via phone, it is lost if not recorded. Emails do not get lost, but can end up in clutter if the user does not organize them systematically. Therefore the application has to store and organize discussion or additional information in a suitable format.

TR600 User Interface

Especially UC300 points out, that the concept must include a user interface. This user interface shall support inspecting and editing instances of the domain model, navigation in the content, dependency visualization, following the decision workflow and collaboration features. Furthermore, a clear terminology has to be established and applied.

Summary of Technical Requirements

Figure 3.6 provides an overview of the functional requirements. The user interface provides access to the four required function blocks dependency management, content repository, decision workflow and collaboration features. The domain model provides the structure for all other requirements, as it models data and behavior of the required application.

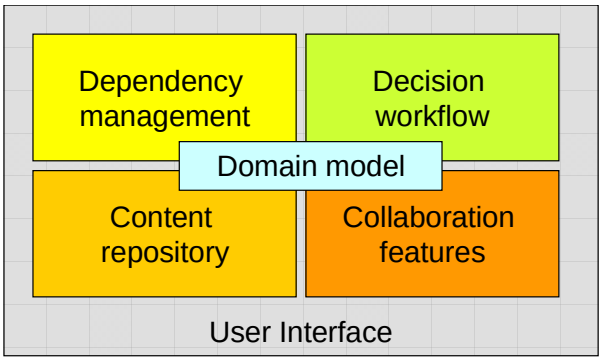


Figure 3.6: Technical requirements for the tool support.

The use cases and the technical function blocks described in this subsection are accompanied by non-functional requirements. These requirements are explained in the following.

3.1.5 Non-functional Requirements and Volume Metrics

Based on the requirements evaluated in previous subsections and the analysis of volume metrics, this subsection outlines the non-functional requirements which support the functional requirements examined in Subsection 3.1.4.

Scalability

Scalability is essential for the usability of the application. Over time an increasing number of architects will work with the application introducing a growing number of assets and workload. Volume metrics retrieved from interviewing the target audience and studying the AD domain allow to expose the expected numbers.

Table 3.1 shows the expected user count, data volume and transactions on the system. One may reckon that all architects use the tool at the same time. The tool will be used by approximately 10 project teams simultaneously. 80% of the decisions are made in the first 20% of a project. After this phase the input continuously decreases. This anticipate peaks in the usage of the tool, especially shortly after the project initiation phase and before deadlines.

Users	
min.	3-5 overall per project
max.	100 overall per project
avg.	10 architects, 40 specialists per project
Data volume for ADs	
min.	50 ADs per project
max.	>1000 ADs per project
avg.	150 ADs per project
Transactions	
read and browse	very often
modify	often
make decision	seldom

Table 3.1: Volume metrics for the tool support.

Storing the content of one AD into a conventional text file shows, that one AD is around 3-5 KByte. On average, this results in less than 1 MByte storage needed overall per project. But due to versioning and additionally uploaded files, the needed storage will increase to several GByte. This increase must be supported by the selected storage.

The application must remain scalable with growing user and data traffic. It must also deal with the mentioned peaks. Response times of page loads must stay in the range of 0-7 seconds. This number results from the fact, that longer response times lead to usability losses as the user's flow is interrupted [33].

Performance

Performance is another important non-functional requirement. As seen in Section 3.1.1, the target audience has to deal with many aspects in a restricted time frame. It is important that the tool supports the architects in helping them to work faster and does not constrain them. Effectiveness of their work depends on fast response time and high throughput. Performance critical components are the storage, the platform on which the application runs (e.g. Web server) and the network. Response times for page loads shall stay below 7 seconds. Memory-intensive transactions like AD content import shall have response times under 15 seconds.

Usability

The target group analysis in Section 3.1.1 points out that architects do not want to spend much time in getting familiar with a new application. As outlined, they have a lot of important things to do; they are under constant time pressure. This means the user interface must be easy to learn and intuitive to use. Thus, a short user's guide or context sensitive help shall support the architects in learning the application. The navigation should be straight forward to use and the application should provide a manageable but sufficient repository of AD descriptions. If the front-end is not easy to learn and use, it probably will not be accepted by the architects.

Table 3.2 shows the expected usability metrics for the most important tasks. These metrics define efficiency, i.e. productivity, errors, and user satisfaction. An indicator for efficiency is the amount of time needed to perform a certain task. This amount should be low. The error rate, measured with the number of mistakes, should be low as well. Whenever a user makes a mistake, he must be able to recover from it easily. Moreover, the user should be satisfied with the overall user interface [33].

	novice user	**expert user**
Import AD content		
time to complete task	60 sec.	15 sec.
number of mistakes	2	0
user's attitude	neutral	positive
Search and inspect specific AD		
time to complete task	60 sec.	15 sec.
number of mistakes	5	1
user's attitude	neutral	positive

Create new AD/edit AD		
time to complete task*	180 sec.	20 sec.
number of mistakes	3	0
user's attitude	neutral	positive
Create discussion for AD		
time to complete task	90 sec.	15 sec.
number of mistakes	2	0
user's attitude	neutral	positive
Document outcome		
time to complete task	180 sec.	15 sec.
number of mistakes	2	0
user's attitude	neutral	positive

*time to type in content not taken into account

Table 3.2: Usability metrics.

Data Integrity

As in every content repository, data integrity is very important. Data integrity also has to be considered when the domain model is mapped to the data source layer.

Maintainability

As one objective of this thesis is the implementation of a prototype and the handing over of it after the completion, the code needs to be maintainable. This requires a clean structuring of code and comments as well as a developer's guide.

Configurability

For now, it is not planned to provide the possibility to configure the prototype.

Documentation

The prototype shall be delivered with the following documentation: installation guide, user's guide, developer's guide. The installation guide provides information how to install the application. The user's guide helps the architects in getting familiar with the most important tasks, provides information about the terminology and terms used in the application, and includes a list of frequently asked questions (FAQs). The developer's guide informs the maintainer of the application about the architecture, the application specifics like used design patterns and assumptions on which the implementation is based.

The analysis of non-functional requirements concludes this section about the requirements for concept and tool. The next section evaluates two candidate frameworks and provides a base for the decision about the underlying platform for the tool-support.

3.2 Analysis of Candidate Assets

The AD 'Platform and Language Preferences' is a key decision in every project. Hence, this decision has to be made in this thesis as well. Another important AD is 'Presentation Layer Technology', i.e. to choose between 'Rich Client', 'Thin Client' or 'Rich Internet Application'. To provide a basis for these decisions in the thesis, this section evaluates candidate assets for the implementation of the application on a conceptual level.

It compares a rich client framework with a Web 2.0 framework. Two promising frameworks are chosen for the comparison: The Eclipse Rich Client Platform (RCP) [36] and the wiki framework QEDWiki [37]. The analysis of these frameworks leads to the decision whether the usage of a framework for the implementation of tool support in this thesis is reasonable. Choosing one of the frameworks decides which platform and language as well as presentation layer technology to use for the implementation. The following subsection outlines the criteria which are used for the comparison.

3.2.1 Criteria for a Comparison of Candidate Assets

For the comparison three criteria are established which are based on the functional and non-functional requirements described in Section 3.1. Criteria which do not play an important role in this thesis are not included although they might be needed in other projects. The criteria are listed in the following.

1. Collaborative content management: How does the framework support content syndication, i.e. the reusage and sharing of content, and content structuring? How does it assure collaborative work on this content through inclusion of e-collaboration features?
2. Usability: How usable are the possible user interfaces created with the help of the framework? How usable is the application with respect to performance and installation?
3. Developer friendliness: How usable and powerful is the extension mechanism? How usable is the framework structure for the developer?

The first two criteria concern the user and administrator of the developed solution. The third criterion addresses the developer who deals with the internal structure of the framework.

The next two subsections provide a short introduction into the architectures of the two candidate frameworks.

3.2.2 Eclipse Rich Client Platform

The Eclipse Rich Client Platform (RCP) [36] is a platform and a framework for developing rich client applications in Java. The Eclipse RCP allows the development of a wide range of rich client applications. This makes the Eclipse RCP a very generic framework for software developers.

The Eclipse RCP includes a core runtime as well as a set of plug-ins. The core runtime is responsible for discovering, loading and executing plug-ins. Any further functionality is implemented in plug-ins.

When capturing the Eclipse RCP into a layered component diagram, one can find that the main focus of RCP is on the presentation layer. Figure 3.7 shows a simplified component diagram of the Eclipse RCP. All components depicted in this figure are plug-ins. The developer of a RCP rich client application can write plug-ins for every layer and interconnect them. In the figure, these plug-ins are represented through the dashed components. Arrows between the components depict communication like function calls.

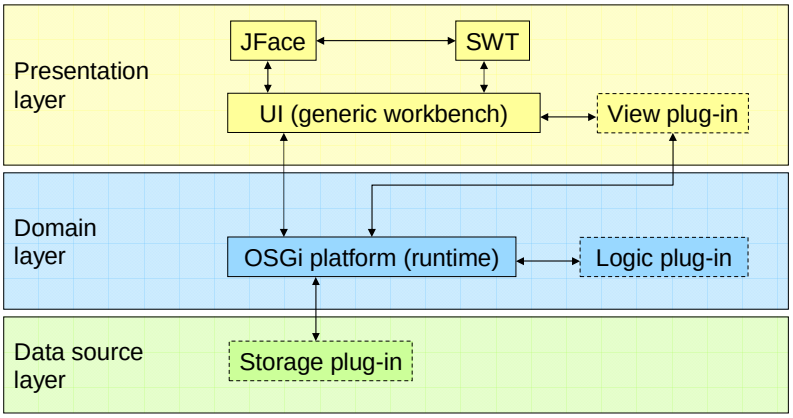


Figure 3.7: A layered view on the Eclipse RCP architecture.

Using the components in the presentation layer, the developer can create powerful user interfaces in Java. The Standard Widget Toolkit (SWT) uses native components of the operating system to provide the look and feel of desktop applications. JFace lays on top of SWT and supports more sophisticated components like wizards, dialogs or the implementation of user interface design patterns. Furthermore, other view components can be integrated using the plug-in mechanism. The Open Services Gateway initiative Alliance (OSGi) platform component on the domain layer implements the OSGi Release 4 specification [38] and builds the core runtime for all other components, which are realized as plug-ins. It acts as a server for the plug-ins, manages their life cycle and provides a registry for them. Via start- and stop-methods a plug-in can be started and stopped without the need to restart the platform. On the domain layer, the developer can include several other domain logic components. The RCP per default does not support a data source layer component. Nevertheless, several products in the form of plug-ins are available to perform the needed functionality. These plug-ins can accomplish persistence, but also communication between client and server or synchronization which can be part of the data source layer.

3.2.3 QEDWiki – Web 2.0 Framework

QEDWiki [37] (short for ‘quick and easily done’ wiki) is a Web 2.0 framework from the IBM Software Group Emerging Technology (version 1.0.0 released on alphaworks in February 2007). It is written in PHP and JavaScript and runs on an Apache Web server. It provides a framework for the construction of Web 2.0 applications. Developers can use QEDWiki to implement a customized wiki whereas users can create simple situational applications without being dependent on the de-

veloper. These users adopt the role of a developer. The wiki content and all situational applications are organized in a wiki structure, i.e. in linked Web pages, which are ordered hierarchically.

QEDWiki is a RIA framework which uses a Web browser for displaying the user interface and a Web server for the logic processing. The Hypertext Transfer Protocol (HTTP) [32] serves as protocol for data communication. The architecture of QEDWiki can be viewed as a 1+3+1 layered architecture, comprising a data tier, an application tier and a client tier. The data tier contains parts of the data source layer (1), the application tier is layered into presentation, domain and data source layer (3) and the client tier includes parts of the presentation layer (1). The Web server comprises data and application tier whereas the Web browser contains the client tier. Figure 3.8 shows a simplified representation of the architecture layers, when a QEDWiki page is loaded in the browser.

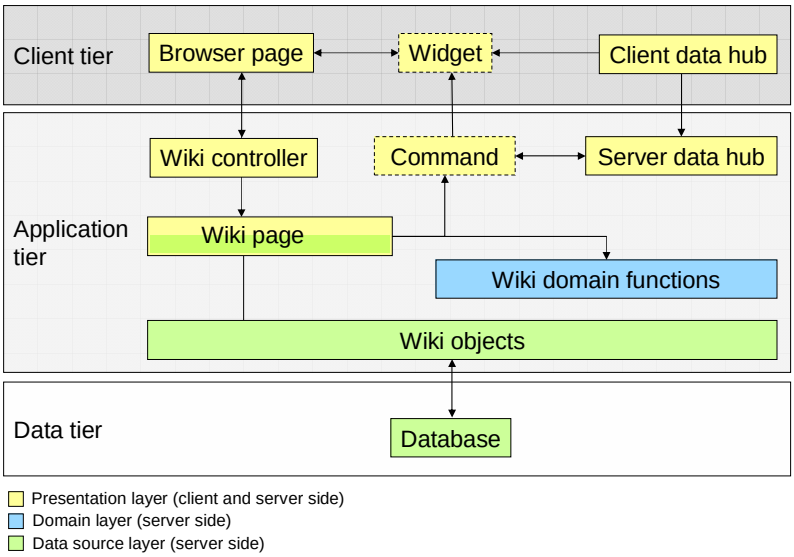


Figure 3.8: A view on the layered architecture of QEDWiki.

The data is stored in the database in the data tier. *Wiki objects* are used as logical data structure and are responsible for the persistence. Examples of wiki objects are user, group or page. The Zend Framework [39] persists these objects in the database. Database and wiki objects belong to the data source layer. On the domain layer, few domain functions exist, which e.g. handle security tasks like authentication. As Figure 3.8 shows, parts of the presentation logic are implemented on the application tier, other parts are implemented on the client tier.

Wiki page is the parent container of extensions to QEDWiki, which are referred to as *commands*. As wiki page is responsible for rendering commands but also is a wiki object, thus responsible for persistence, no clear assignment to either the presentation or the data source layer can be made. Commands are located on the presentation layer and can implement any functionality which is then included into the page at rendering time. They are written in PHP but may also contain JavaScript elements which are interpreted on the client at runtime. The rendered instance of a command on the presentation layer is called *widget*. This means, a widget only exists if a page is executed which contains it. As a widget can include JavaScript code elements which are interpreted at runtime, it can provide active elements for the user. The *wiki controller* is responsible for choosing the appropriate page for displaying.

Through the *data hub*, widgets can communicate with other widgets and commands, which are included in the actual page. Figure 3.9 depicts the functionality of the QEDWiki data hub with an example which is taken and extended from the QEDWiki developer's guide. A page includes two commands, which results in two widgets on the client side. The first widget, the AddressList, contains several addresses which can be selected in the user interface. The other widget, GoogleMaps, displays a given address in a map. These two widgets should now be connected in a way that the selected address in the AddressList widget is shown in the GoogleMaps widget.

The data hub is distributed over the client and the server and connected via Ajax (see Section 2.5). Data is exchanged using the JavaScript Object Notation (JSON) [40]. The functionality of the server data hub is implemented in PHP, whereas the data hub on the client uses JavaScript.

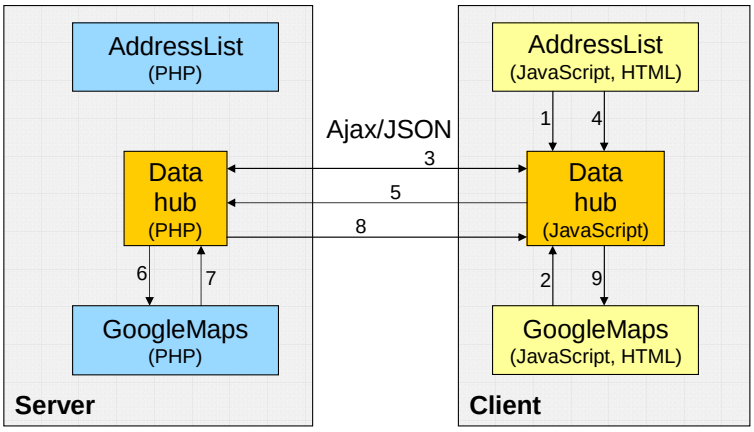


Figure 3.9: The QEDWiki data hub functionality.

The data hub uses the publish-subscribe pattern [41]. The AddressList widget registers itself as publisher on the data hub (1), whereas the GoogleMaps widget registers as subscriber on the data hub (2). Client and server data hub exchange this information (3). The registration is stored persistently in the data storage on the server. If the user selects an entry in the AddressList widget, the selected data and widget information is sent to the client data hub (4) which it redirects to the server data hub via Ajax (5). The server data hub then asks all subscribers of this data to re-render, in this case the GoogleMaps command (6). The rendered results (7) are then returned to the client data hub (8) which updates the respective GoogleMaps section on the page (9).

The data hub is especially interesting for users developing situational applications. They can assemble different widgets on a page and connect them via the data hub. These widgets can also connect to sources outside the wiki, e.g. feeds or blogs.

After this short introduction into the Eclipse RCP and QEDWiki, the next subsection compares both frameworks using the criteria established in Subsection 3.2.1.

3.2.4 Comparison of Eclipse Rich Client Platform and QEDWiki

As explained in the previous subsections, the Eclipse RCP represents a typical rich client framework whereas QEDWiki is a RIA framework. Figure 3.10 shows how client and server of QEDWiki and Eclipse RCP are distributed on the three logical layers presentation, domain and data source.

The presentation logic of the Eclipse RCP application is installed on the client. In the figure, even the domain logic is located on the client. The Eclipse RCP allows the developer to decide where the domain logic should run, it even can be split. The client in the figure also contains a data storage device. The shared data is located on the server.

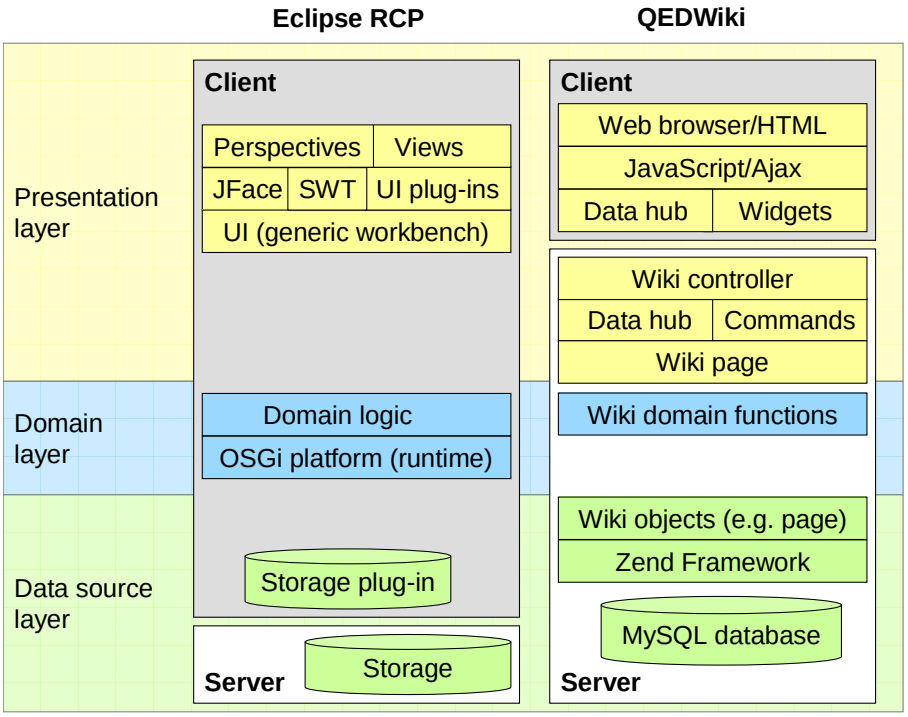


Figure 3.10: Comparison of RIA and rich client using QEDWiki and Eclipse RCP as an example.

QEDWiki as RIA framework has a very lightweight client, which only contains parts of the presentation layer. QEDWiki is not a pure thin client application because parts of the presentation logic are implemented on the client in JavaScript. This is typical for newer Web 2.0 applications. The server includes parts of the presentation logic, the domain logic and the data source.

The following deepens the comparison between QEDWiki and Eclipse RCP based on the criteria established in Subsection refsec:evalcriteria. During this comparison, details of the frameworks are explored but also the differences between rich clients and RIAs are elaborated upon.

Collaborative Content Management

Table 3.3 describes the differences between the Eclipse RCP and QEDWiki concerning the criterion collaborative content management defined in Subsection 3.2.1.

Criterion	Eclipse RCP	QEDWiki
Content storage	The Eclipse RCP per default does not contain a resource model, therefore the developer can use any data stores he wants, be it a database or file system. Several resource model plug-ins are available for extending the core.	The flexibility of the content repository design in QEDWiki is restricted, since it already provides a MySQL database for data storage. The MySQL database stores data included in wiki pages. Other files related to pages can be stored on the Apache Web server.

Content sharing/syndication	The sharing of content in an Eclipse RCP application can be accomplished through a composition of plug-ins. Choosing a content versioning plug-in like Concurrent Versions System (CVS) [42] for instance can help sharing and tracking changes in content which is stored in files. However, these implementations typically create local copies of the file for each user, which can lead to users working on different versions of one file. Relational database management systems like Apache Derby [43] provide databases which can be installed locally or on a remote server. Derby provides a plug-in for the Eclipse RCP which can run in different modes: Users can work on shared data on the remote database (client/server mode) or locally on their own copy (embedded mode). The problem of inconsistency between server and client data can be solved by the implementation of synchronization mechanisms.	QEDWiki already provides a wiki and consequently a shared content repository. All users work on the same instance of content, there exist no local copies of content. Versioning of pages allows tracking changes and associating them to a particular user. Locking mechanisms prevent inconsistencies and the simultaneous editing of content through two or more users.
Collaborative work	Collaboration features can be included into the Eclipse RCP using plug-ins. JAZZ [44] for instance enhances some of the existing plug-ins and provides an Eclipse-based collaboration platform for software developers.	QEDWiki per default provides collaboration features like a message board for each wiki page. In contrast to Eclipse RCP, user management is already implemented and used for the collaboration.
Structured content and data model	The design and implementation of the data model and structured content is left to the developer.	QEDWiki already implements a data model which can be extended to the own needs. As every wiki page may have children, the content is structured in a tree. Content on pages can be structured as well.

Table 3.3: Comparison of Eclipse RCP and QEDWiki under criterion collaborative content management.

The scope of Eclipse RCP is more general than the one of QEDWiki. The plug-in concept allows the developer a great flexibility in the design of his application. On the other hand this also means more decisions become necessary, and more mistakes can be made. The realization of a collaborative content repository in Eclipse requires the thoughtful assembly of plug-ins. QEDWiki is more specialized and dictates several components like data model or storage device. On the other hand it provides features needed to implement a collaborative content repository, e.g. collaboration features and user management.

The analysis of the two frameworks with respect to collaborative content management exposes differences which can be generalized to rich and thin client/RIA architectures. First of all rich clients can support redundant data storage on the client, which allows the user to work with the content even if the server is not reachable. Rich clients which implement this feature have to face inconsistencies or users working on different copies of a document. In thin client applications/RIAs it is not possible to reach the content if the server is not available. Users of thin clients/RIAs all work on the same instance of the data, changes are updated immediately. This reveals the problem of concurrent access which for instance can be solved by locking.

Usability

Table 3.4 describes the differences between the Eclipse RCP and QEDWiki concerning the second criterion defined in Subsection 3.2.1: usability.

Criterion	Eclipse RCP	QEDWiki
User interface development	For the development of user interfaces, besides SWT or JFace toolkits like the Forms API [45] (pages 164ff.) can be integrated into the rich client application. The Forms API supports HTML-like forms which can seamlessly be integrated into the application.	The client user interface of QEDWiki applications is displayed in a Web browser, i.e. the technologies used are HTML, CSS and JavaScript. The user interface can be adapted by the user through widgets.
Usability of interface	The usability of the interface depends on the custom design. Often the perspective and view mechanisms are used for the implementation of the user interface. This ensures the consistent look and feel of Eclipse RCP applications, but is not always the best solution.	The QEDWiki user interface already implements several best practices in Web site usability, like displaying the navigation on the left hand side. A first test showed, that the user interface is good, but improvable. To improve the usability, the developer can adapt the user interface.

Table 3.4: Comparison of Eclipse RCP and QEDWiki under criterion usability.

The Eclipse RCP leaves the design of the user interface to the developer. This gives a great flexibility in the design, but, like in the design of architecture, many decisions have to be made. QEDWiki provides a complete wiki user interface which can be adapted. This again restricts to a certain user interface, but also provides an entry point for the user interface developer.

The analysis of the frameworks with respect to usability exposes several differences between rich and thin clients/RIAs. Depending on the user interface design, rich clients can provide highly usable applications with nearly any functionality. A thin client, e.g. Web browser user interface in general is not as sophisticated as a rich client user interface. Web 2.0 RIAs like QEDWiki address this shortcoming by including JavaScript libraries to enrich the user interface with interactive features. RIAs seem to overcome the gap between the powerful user interfaces of rich clients and the simple page-by-page mechanism of thin clients.

The client part of rich client applications has to be installed on the client machine. As thin clients/RIAs often use Web browsers for their user interface, the client part is already installed.

The logic processing of rich client applications is distributed on client and server, thus application performance can be improved. As clients are never the same, performance measures on the clients may differ. The thin client processing is done on the server. This means, the whole performance depends on the server. This can lead to bottlenecks, e.g. if many persons are online at the same time.

Developer Friendliness

Table 3.5 describes the comparison of the Eclipse RCP and QEDWiki with respect to the third criterion defined in Subsection 3.2.1: developer friendliness.

Criterion	Eclipse RCP	QEDWiki
Extension mechanism	Eclipse RCP is based on its plug-in mechanism. Nearly anything can be included as plug-in. If a functionality is not available from other sources, the developer can create a new plug-in. Getting started with the plug-in mechanism is not trivial, since the mechanism is very powerful. But Eclipse RCP supports a user interface for the creation of rich client applications and plug-ins.	The extension mechanism of QEDWiki addresses both the developer and the user of the application. The developer implements functionality which can then be assembled by the user. Widgets are mainly used to prepare data for display, this means they are located on the presentation layer. The creation of widgets requires the extension of an existing wiki class and the insertion of this extension into the right directory on the server.
Framework structure	Developing plug-ins for an Eclipse RCP application is difficult, because many dependencies between different plug-ins exist. The developer has to understand how to use the extension points of the various plug-ins. Furthermore several relicts from old Eclipse RCP versions exist, which can confuse the developer.	The naming of the components, files and functions is reasonable and thus easy to understand. QEDWiki uses object-oriented PHP. The extension mechanism is easy to use.
Documentation and community	There is a big Eclipse RCP community writing plug-ins and tutorials for the framework. Furthermore there are several good books on the market. This eases the entry into the plug-in development.	The extension mechanism is straightforward, but QEDWiki is not yet documented in detail and lacks a big community, since it is in an early development state. This makes the usage of QEDWiki functionality difficult.

Table 3.5: Comparison of Eclipse RCP and QEDWiki under criterion developer friendliness.

In contrast to Eclipse RCP, QEDWiki provides an extension mechanism for both, developer and user. This is influenced by the Web 2.0 paradigm, which is more user-centric than the conventional application paradigm.

Eclipse RCP provides a powerful extension mechanism, which is well documented. However, due to relicts of previous versions and the growing number of dependencies between plug-ins when developing sophisticated applications, the extension mechanism is hard to use. The extension mechanism of QEDWiki is easier to use but hard to learn, because the first version of QEDWiki was only released recently. Thus, its user and developer community is still small.

The analysis of the frameworks with respect to developer friendliness exposes differences and similarities between rich client and RIA frameworks in general. Since the history of RIA frameworks is shorter than the history of rich client frameworks, they do not yet have have the same stage of maturation as rich client frameworks and they still lack large communities. However, their architecture and code structure is comparable to rich client frameworks. For the implementation of RIA frameworks, developers can use best practices and architecture knowledge on frameworks grown during the development of rich client frameworks. This means, application developers who are familiar with rich client frameworks will be able to adapt their knowledge for application development with RIA frameworks. This can lead to better acceptance of RIA frameworks through developers.

3.2.5 Summary

Eclipse RCP and QEDWiki are both promising candidates to implement collaborative decision making. Unlike QEDWiki, Eclipse RCP is very generic. This allows the developer great flexibility in the design of his application and the choice of the plug-ins. However, the free choice of components does not ensure a good architecture. The components must harmonize and all needed features must be implemented or at least plugged-in. QEDWiki in contrast, implements most of the needed functionality and provides an extension mechanism which can be used to implement the missing features. It is a very specialized framework which does not have a big community yet.

The examination of QEDWiki framework makes clear that RIA frameworks can be compared to rich client frameworks: They both can support a layered architecture, an extension mechanism and a rich user experience.

The choice of one of the two frameworks decides the AD ‘Platform and Language Preferences’: Choosing the Eclipse RCP as underlying framework selects alternative ‘J(2)EE and Java’ whereas the choice of QEDWiki selects alternative ‘Linux, Apache, MySQL, PHP (LAMP)’. The AD ‘Presentation Layer Technology’ is decided as well: Choosing Eclipse RCP decides for a ‘Rich Client’, QEDWiki for a ‘Rich Internet Application/Web 2.0’.

Which alternative is chosen for the application design in this thesis is described in the following Chapter 4, the conceptual design of the system.

4 Conceptual Design: AD_{kwik}

The previous chapter analyzes the requirements and candidate assets for the realization of a tool-supported concept and prototype for collaborative AD modeling, making and knowledge management. This chapter presents the conceptual design to meet these requirements: *AD_{kwik}* (pronounced "AD-quick"), which stands for Architectural Decision knowledge Web integration kit. The conceptual design includes the architecture of AD_{kwik}, the suggested domain model and methods for the realization of dependency management and decision workflow. Furthermore, it suggests assets and patterns suited to realize the content repository, collaboration features and a user interface.

4.1 Architecture Overview of AD_{kwik}

Considering the supported features of QEDWiki outlined in Section 3.2, the usage of QEDWiki as framework for the implementation is reasonable: As QEDWiki supports a certain type of applications, namely wikis, it is more suitable than the generic Eclipse RCP. On the one hand, the usage of QEDWiki as base for the implementation pre-defines the architecture to design. On the other hand it provides several important features which are needed in AD_{kwik}, thus do not have to be implemented from scratch. QEDWiki already implements user management and collaboration features like a message board. It inherently is a content repository, providing data storage in a database, simple versioning of pages and a search functionality. As RIA, QEDWiki supports a rich user interface, but enables easy access to the application through a Web browser as well as suitable sharing of content. The extension mechanism of QEDWiki is not as powerful as the extension mechanism of the Eclipse RCP. Nevertheless, it is sufficient for the implementation of the required features like domain model, decision workflow and dependency management. Taking these advantages of QEDWiki as justification, the AD 'Platform and Language Preferences' in this thesis is made in favor of QEDWiki, hence, the alternative 'Linux, Apache, MySQL, PHP (LAMP)' is chosen. The AD 'Presentation Layer Technology' is decided for 'Rich Internet Application'.

The architecture of AD_{kwik} is made up of three logical layers, using the QEDWiki framework layering model. An abstract component model of the layered AD_{kwik} architecture is depicted in Figure 4.1. It includes the following layers:

- **Data source layer** – The ADkwikDataSourceLayer on the left side of the figure provides an interface for direct operations on the data, e.g. saving, updating and deleting from the database. Furthermore, it implements the domain model in the AdDataSourceComponent and the persistence in the StorageDataSourceComponent. It is responsible for the correct storage of data. The domain layer components use it as data repository for their computations.

- **Domain layer** – The ADkwikDomainLayer in the middle contains business logic of the AD domain and is responsible for the correct modification of content. It provides interfaces to create, edit and delete AD content regarding data integrity and versioning. Furthermore, it provides interfaces and implementations for dependency management, decision workflow and the pre-population with decision content through import. Technical components in the domain layer are the LoggingDomainComponent, the ErrorHandlingDomainComponent and the SecurityDomainComponent, which handles authentication and authorization. The ADkwikDomainLayer receives requests from the ADkwikPresentationLayer, processes them and provides the calculated results back to the ADkwikPresentationLayer.
- **Presentation layer** – The ADkwikPresentationLayer on the right side in the figure includes different presentation components, which are views for the users to interact with the application. The views provide the user interface to modify content, navigate or follow the processes which are provided through the domain layer. Moreover, the ADkwikPresentationLayer contains the CollaborationComponent, which includes collaboration features, and the UIControllerComponent. The UIControllerComponent is responsible for delegating user requests from the different presentation components to the correct domain layer component. It is also responsible to delegate the appropriate view to the user.

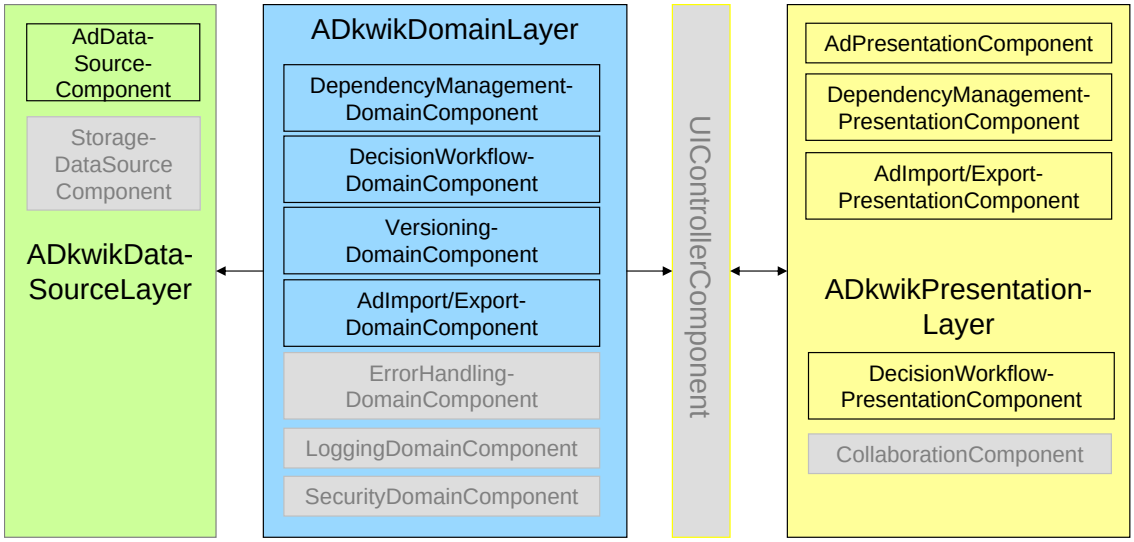


Figure 4.1: Component model of AD_{kwik}. QEDWiki components are depicted in gray.

All components depicted in the figure are relevant to fulfill the functional and non-functional requirements outlined in Chapter 3. Nevertheless, not all have to be implemented, as the QEDWiki framework already is shipped with several components. These components are depicted in gray. The architecture of QEDWiki was explained in Section 3.2.3. Thus, QEDWiki architecture and concepts will be used in the following without further explanation.

The six technical requirements summarized in Figure 3.6 on page 29 are jointly addressed by several components and layers. The following sections introduce concepts, algorithms and models how to realize these requirements and how they can be mapped to the component model.

4.2 Domain Model

As outlined in Section 3.1.4 a domain model for ADs is required. The proposed domain model is a combination of elements of research work [4, 12], methods like the IGSM [14], tools like the AWB [13], and experience of users.

4.2.1 Domain Model Classes and Attributes

ADs are structured in a *decision tree*. As depicted in Figure 4.2 the decision tree is built through the classes *AdLevel* and *AdTopic*. These classes represent *AD levels* and *AD topics*, respectively, which build structuring containers for ADs in the decision tree. AD levels can contain zero or more AD topics, but one AD topic can only have one parent AD level in which it is included. This is depicted with the multiplicity 1 at the upper end of the association between the AdTopic and AdLevel classes in the figure. Analogous, an AD topic can contain other AD topics or have an AD topic as parent. However, the parent of an AD topic is either of type AdLevel or of type AdTopic. This is depicted with the constraint attached to the AdTopic class. One AD can only be included in one AD topic, but an AD topic can include several ADs. Using these multiplicities, only directed graph structures are allowed, ensuring that the decisions are ordered in trees. An AD level is the root of a decision tree.

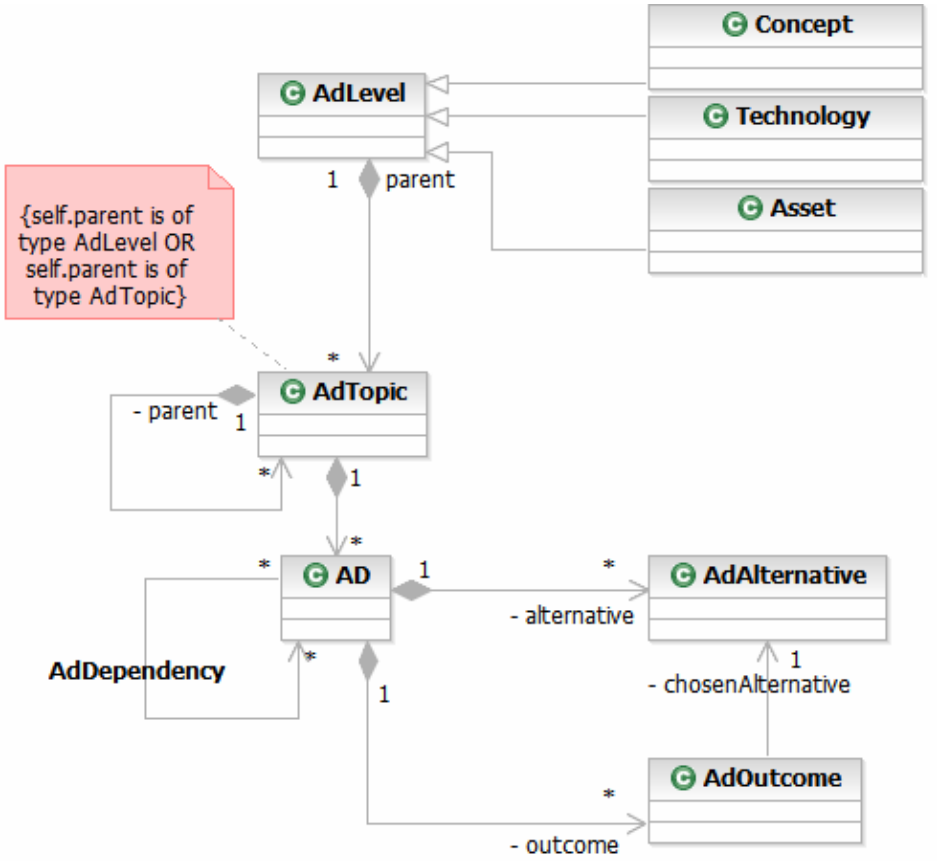


Figure 4.2: Domain model including decision tree, modeled in UML [3].

In the example of SOA, AD levels reflect the established development process, which includes the AD levels *concept*, *technology* and *asset*.

- **Concept** – This AD level contains strategical/conceptual ADs. These ADs are about abstract principles and patterns, which are platform independent. This AD level therefore is also called platform independent model (PIM) [46]. The decisions in this AD level usually are made in the early phase of a project, e.g. the inception phase in RUP [2] or the solution outline phase in IBM Global Services Method (IGSM) [35], and comprise global decisions and abstract views on process and service realization decisions.

  Examples: 'Architectural Style': In the scenario the architects decided to use SOA. 'Platform and Language Preferences': Alternatives are 'J(2)EE and Java', '.NET and C#', 'Linux, Apache, MySQL, PHP (LAMP)' and many more.

- **Technology** – This AD level contains technology decisions, but not yet decisions about specific products or implementations. It is also called PIM/platform specific model (PSM) [46]. Decisions in this AD level concern the design of the overall architecture, i.e. are made in the elaboration phase of RUP or macro design phase of IGSM.

  Example: the 'Presentation Layer Technology'. Alternatives would be 'Thin Client', 'Rich Client' or 'Rich Internet Application/Web 2.0'.

- **Asset** – This AD level contains ADs about products or open source software. The ADs in this layer are platform specific. Therefore, it is also called PSM. ADs in this AD level turn the architecture and design into implementation specific views and are taken during phases like the construction phase in RUP or micro design and build cycle phases of IGSM.

  Example: Which framework to use for the implementation of the system. Alternatives are for example 'Eclipse RCP' and 'QEDWiki'.

Figure 4.3 shows the expanded entities *AD*, *AdAlternative* and *AdOutcome* and their attributes.

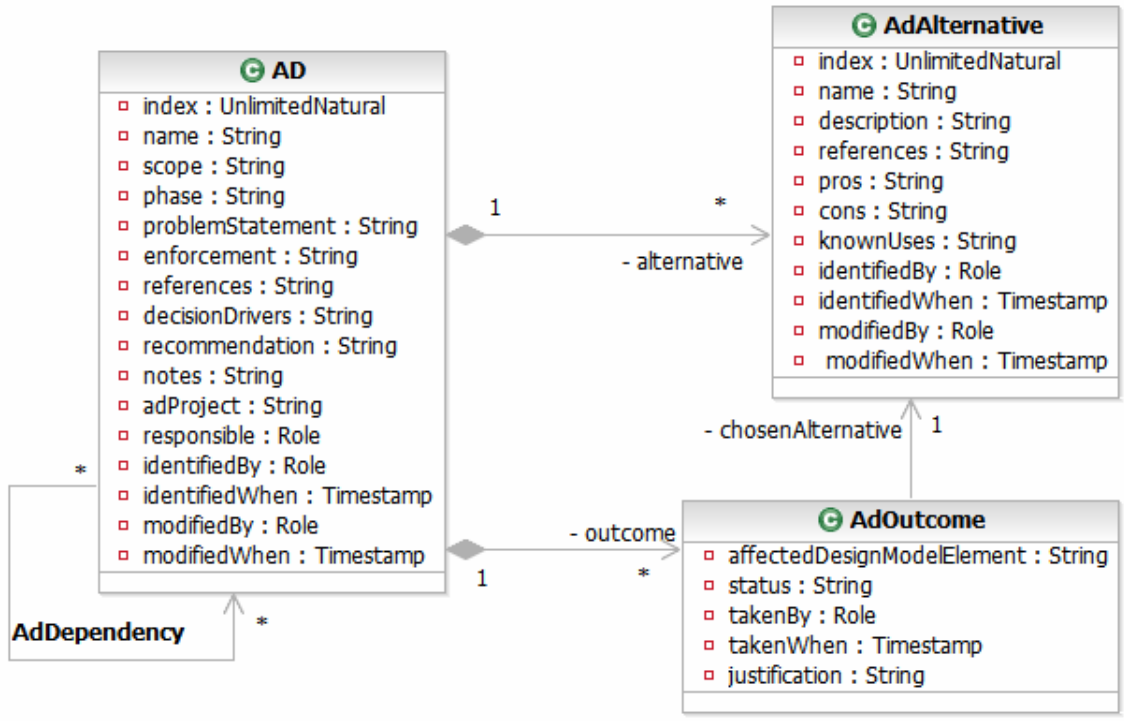


Figure 4.3: Expanded extract of the domain model in UML [3].

The central part of the domain model is the class AD which contains several attributes. The attributes of the class are called *AD attributes* and represent meaningful aspects of an AD. These attributes are explained in the following.

- Every AD has a *name*. An example AD on the conceptual AD level has the name 'Presentation Layer Technology'.
- Furthermore it has an *index*, which is a unique identifier. The unique identifier of the AD with name 'Presentation Layer Technology' is defined here as '15'.
- Attribute *scope* captures the type of the element which the AD covers in the architecture design model, e.g. 'service' if the AD is a decision about the design of a service. An example is the AD 'Transport Binding' which relates to service design, thus its scope is 'service'. AD 'Presentation Layer Technology' is a global decision which does not relate to a specific element in the architecture design model, i.e. its scope is 'global'.
- *Phase* denotes in which phase of a project the AD has to be decided. The decision about the 'Presentation Layer Technology' is made in phase 'solution outline'. Solution outline is the term used in the IGSM to specify the early phase in a project. In this phase high-level requirements are gathered and an initial architecture is defined to comprehend the outline of the system to design [35]. Other phases are for instance 'macro design' or 'micro design'.
- The *problemStatement* describes what the decision is about. The problem statement of the 'Presentation Layer Technology' is 'Users of an application need to be able to assess and manage the application through a user interface. Without a suitable presentation layer technology, applications will not be able to deliver much business value.'.
- *Enforcement* is the choice of enforcing the decision in the development manually, e.g. through coaching or code reviews, or by transformation with tools, e.g. generation from specifications into code [8].
- *References* contain links to external resources like books, papers or Web pages. These resources provide further information about the decision context.
- *DecisionDrivers* offer the possibility to capture non-functional requirements, constraints or quality factors which may influence the AD or the outcome of the AD. As often personal experiences and preferences are the main decision drivers and strategic decisions often negatively influence the decision making, this attribute captures objective information about forces influencing the decision [10].
- The *recommendation* attribute captures recommendations concerning which alternative to choose.
- In the *notes* attribute unstructured information or information about other attribute values can be captured.
- The attribute *adProject* contains the name of the project to which the AD belongs.
- Every AD in a project has a *responsible* role.
- Each AD is identified by (*identifiedBy*) a person at a particular time (*identifiedWhen*).
- Modifications of an AD are captured with the attributes *modifiedBy* and *modifiedWhen*.

Figure 4.3 also shows the relationships between ADs and the classes AdAlternative and AdOutcome. Each AD contains none, one, or more AdAlternatives. In addition it includes none, one, or more AdOutcomes.

An alternative is captured through several elements. Analogous to the AD, the AdAlternative class contains an *index*, a meaningful *name*, *references*, and the attributes *identifiedBy*, *identifiedWhen*, *modifiedBy* and *modifiedWhen*. Example alternatives for 'Presentation Layer Technology' could have the name 'Rich Client' or 'Thin Client'. The *description* element shortly explains the alternative. *Pros* and *cons* contain advantages and disadvantages of the alternative and provide means to evaluate it. Finally, *knownUses* help to understand how the alternative was applied in similar projects.

In some cases, the same AD has to be made for several design model elements of the system. If for example a business process has to be implemented as a set of composed Web services, for each implemented Web service the AD about the 'Transport Binding' protocol has to be made. Alternatives of this AD are 'SOAP/HTTP' and 'Reliable transport, e.g. JMS'. AD attributes like problemStatement, decisionDrivers and references remain the same, whereas the AdOutcome attributes *chosenAlternative* and *justification* for this alternative may differ [10]. In these cases, an AD references several AdOutcomes, e.g. one for each Web service. The attribute *affectedDesignModelElement* in AdOutcome captures the design model element the AdOutcome covers in the system design. Several ADs have to be made once for a system, i.e. they reference only one AdOutcome. An example is the AD 'Presentation Layer Technology'. The Attribute affectedDesignModelElement for its sole outcome is 'Overall Software Architecture'. As these ADs affect the whole design, they have the scope 'global'. AdOutcome attribute *status* shows in which state the outcome is: Is it decided or not? At the beginning of the project, the AD with index '15' has an AdOutcome in state 'open'. *TakenBy* and *takenWhen* capture information about who decided when.

The domain model is represented in AD_{kwik} in different components. The ADkwikPresentationLayer includes several views showing elements of the domain model. Furthermore, the domain model is mapped onto the database scheme which is represented through the AdDataSourceComponent in the data source layer.

The association with the name *AdDependency* depicted in Figure 4.3 denotes relationships between ADs. This dependency association is explained in the following.

4.2.2 Dependency Management

The capturing and usage of dependencies between ADs offer a wide range of applications in AD_{kwik}. Various types of dependencies can exist between different elements of the domain model. In AD_{kwik}, three useful types of dependencies can be identified which are explained in the following including their various influences.

- **Topic dependency** – A *topic dependency* denotes a relationship between ADs whose content belongs to the same AD topic. Typically these ADs are associated with the same AD

topic in the decision tree. A topic dependency is an intra-level dependency, i.e. it only exists between ADs which are in the same AD level. It influences the order in the decision tree. This dependency is of structuring character; it has no direct influence on an AD or its outcome. Figure 4.4 shows several dependent ADs; the arrows depict dependencies of ADs which belong to the same AD topic.

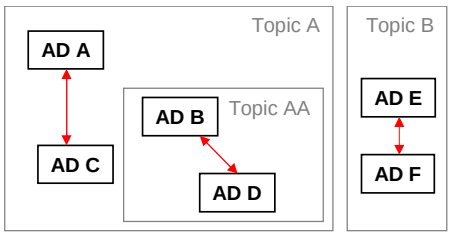


Figure 4.4: Topic dependencies between ADs.

Example: AD topic 'Technology and Process Decisions' comprises ADs which require executive level attention and have a high impact on all other ADs. To these ADs belongs the 'Method Selection', i.e. which methodology is used in the project. Alternatives are e.g. 'Rational Unified Process (RUP)' or 'IBM Global Services Method (IGSM)'. Another AD in the same topic is 'Platform and Language Preferences'.

- **Time dependency** – A *time dependency* denotes a relationship between two ADs which are decided in a certain order. Time dependencies can be inter- but also intra-level. If applied consistently throughout the decision tree, these dependencies influence the sequence of ADs which have to be decided (see Figure 4.5). Time dependencies can be used to provide a guidance for the architect through the decision making process. After every decision, the architect could be informed about the decisions that have to be made next.

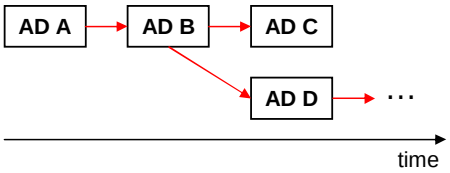


Figure 4.5: Time dependencies between ADs.

Example: The AD 'Method Selection' in the concept level, should be decided prior to all ADs in the technology and asset levels, since it influences the way in which the system design evolves and the ADs are made.

- **Outcome dependency** – An *outcome dependency* denotes a relationship between an AD and another AD or its alternatives. There are two different types of outcome dependencies. The first row of Table 4.1 depicts the outcome dependency between a chosen alternative and an AD, whereas the second row shows the outcome dependency between alternatives of two ADs.

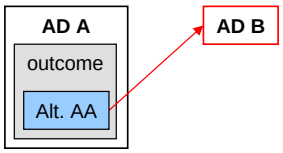
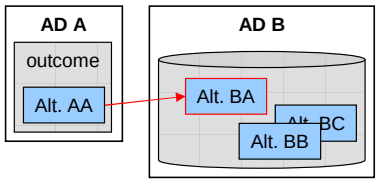
Type of outcome dependency	Explanation
	AD A is decided in favor of alternative AA. This influences AD B. Example: AD 'Platform and Language Preferences' (AD A in the figure) is decided for alternative 'Linux, Apache, MySQL, PHP (LAMP)' (alternative AA), AD 'Java JDK' (AD B) becomes obsolete.
	AD A is decided in favor of alternative AA. This influences one or more alternatives in the alternative pool of AD B. If only one possible alternative is left, it could even influence the state of the dependent AD B: The decision is pre-decided by the first one. Example: AD 'Task Invocation Interface' (AD A in the figure) on the concept AD level is decided in favor of 'Web application, page flow', then alternative 'Rich Client' (alternative BA) of 'Presentation Layer Technology' (AD B) in the technology level becomes obsolete.

Table 4.1: Types of outcome dependencies.

Outcome dependencies can be used for revealing influences of a decision on single ADs or the whole decision tree to the point of what-if simulations. Another application is guidance through the decision making process by masking irrelevant ADs or alternatives.

Time and outcome dependencies can have different *influence types*, which can exist between ADs as well as ADs and alternatives. The following influence types were identified by the target audience through the analysis of sample AD content:

- *influences*: This influence type expresses a generic weak dependency. It is bidirectional. Dependent ADs have similar decision drivers. This dependency is default.
- *forbids*: If AD A is decided for a certain alternative, AD B must be dropped. An example is given in the first row of Table 4.1: Alternative 'Linux, Apache, MySQL, PHP (LAMP)' forbids AD 'Java JDK', i.e. AD 'Java JDK' becomes obsolete.
- *constrains*: This type expresses a weaker form of forbids. An example is given in the second row of Table 4.1: AD 'Task Invocation Interface' constrains 'Presentation Layer Technology' as one of its alternatives becomes obsolete.
- *refines*: This influence type describes dependencies between ADs which are located in different AD levels, i.e. concept, technology or asset. ADs on the technology level refine ADs on the concept level, whereas ADs on the asset level refine ADs on the technology level. Based on decisions of the AD level above, a more detailed AD concerning one aspect of the system is made; this is called refinement. The AD 'Presentation Layer Technology' on the technology level refines the AD 'Task Invocation Interface' on the concept level.
- *triggers/leads to*: As soon as AD A is decided, AD B can be decided as well. This influence type is used for time dependencies.

To realize topic dependencies, the ADs are structured in the decision tree. Time and outcome dependencies are added to the ADs as supplemental element. This element contains the influence type and – if necessary – the influencing/influenced alternative. The dependency management is

distributed over all three architecture layers including a user interface to manage the dependencies, a dependency logic in the DependencyManagementDomainComponent and the storage of dependencies in the database through the AdDataSourceComponent and the StorageDataSourceComponent.

The content repository to store the dependencies as well as ADs, alternatives and outcomes is explained in the following section.

4.3 Content Repository

As shown in Figure 4.6 the content repository (TR300) of AD_{kwik} consists of a database and a file system. The file system is used for additional assets like papers, code or images, whereas the database persists entities structured according to the domain model (TR310). The use of a database supports versioning and search ability on attribute level. If stored in files, only the versioning of the whole description is possible. Search functionality would parse all files.

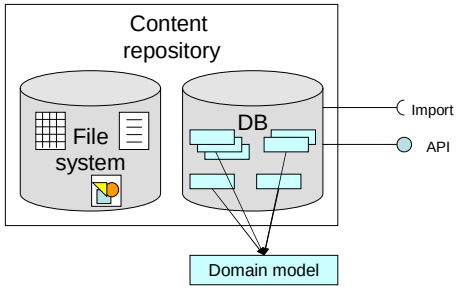


Figure 4.6: The content repository of AD_{kwik} consisting of a database and a file system.

For the storage of data into the database, i.e. the object-relational mapping in the AdDataSourceComponent, the Active Record pattern is used. According to Fowler [19], an Active Record is an "object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data". For each column in the database table, an Active Record class contains one attribute. Furthermore, it includes logic to access the data, i.e. storing and updating it in the database. The implemented domain logic can cover validations or simple computations. This means, an Active Record strictly speaking addresses both, the data source and the domain layer.

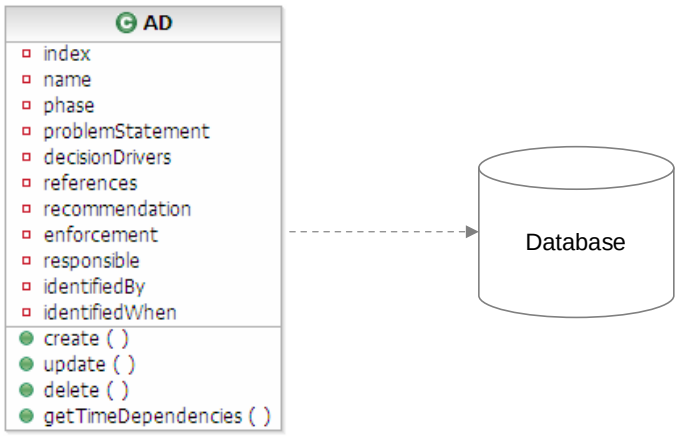


Figure 4.7: Active Record pattern example.

Figure 4.7 shows the class AD as Active Record example. The domain model attributes of the class map exactly to the columns of the database table for AD records. The Active Record provides create, update and delete functions, as well as the simple domain logic function getTimeDependencies(). These functions can also include versioning mechanisms.

The benefit of the Active Record pattern is that there is no need to code SQL directly, since it is encapsulated in the Active Record. Hence, it is straightforward to use.

To implement versioning (TR320) in the VersioningDomainComponent, the Edition design pattern is used. Figure 4.8 shows an overview of the pattern.

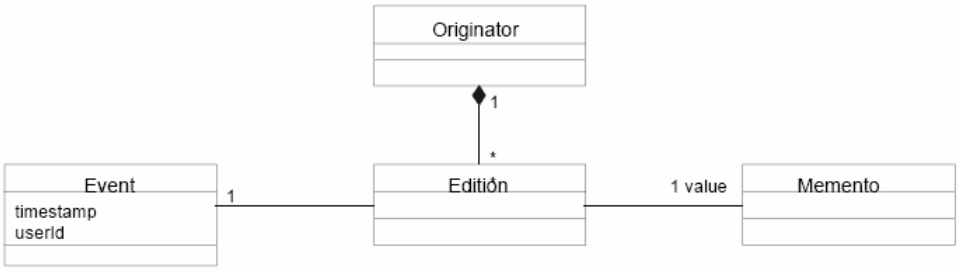


Figure 4.8: Edition pattern for versioning. [47]

The versioning is based on the association of the state of a database record with the event that caused the change [47]. The *originator* is an object with a state. In this pattern, the originator is not only the creator of an *edition* but also the caretaker. This means, it is not only responsible for creating new editions, but also has to manage them. *Memento* is the value of an edition whereas *event* is the key. An event contains the action which caused the creation of the edition.

To give an example in the AD domain on the database level, the originator is one AD record in the database. Whenever an event occurs, e.g. the AD is updated because a user changed an attribute value, a new edition, i.e. database record of the AD is created from the originator. This can happen through cloning the originator database record and assigning the changed attribute values to it. The attribute values of the AD build the new memento of the edition. In addition, to differentiate the editions, a database record for the event is created and associated with the new edition as key. The edition with the latest timestamp in the event is the most actual AD.

Using this pattern in AD_{kwik} allows tracking who changed what at which time based on events, and the rollback to an earlier edition. Moreover, the states of several editions can be compared. When using one database record for each edition, this comparison can even be conducted on attribute level.

As the storage shall provide an API (TR330) to use the content also in other applications, a simple API which also supports the versioning mechanism is part of the concept. A Web service interface can be created on top of it.

To pre-populate the application with decision content, TR340 requires an import mechanism. Figure 4.9 shows the layered component model for the import/export mechanism.

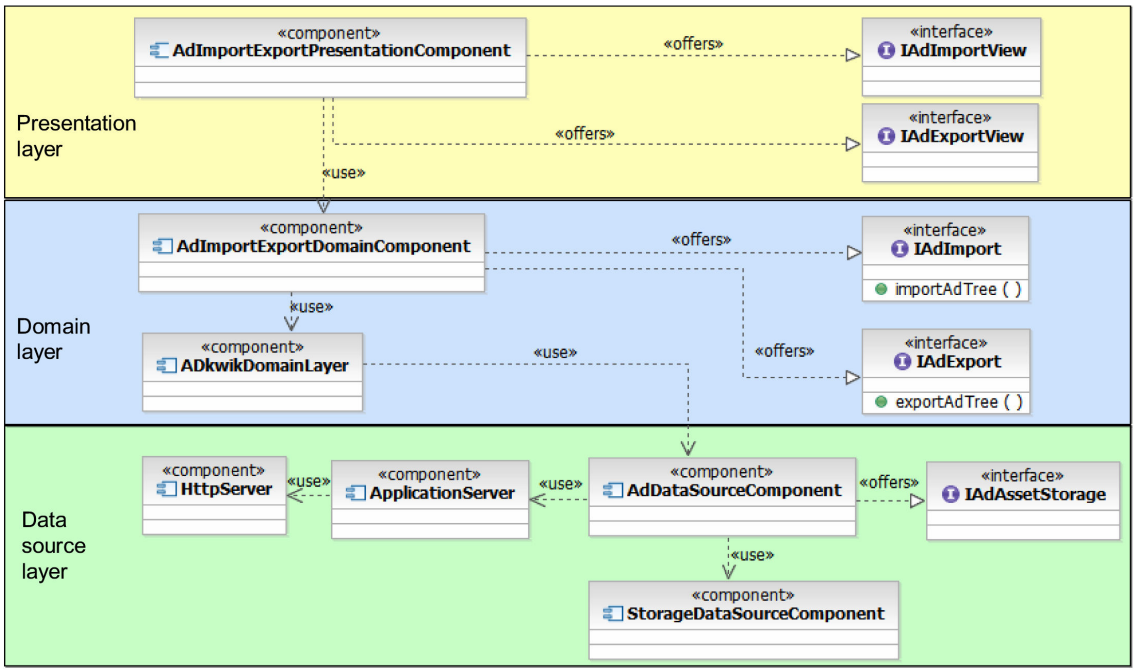


Figure 4.9: Layered component model for the import/export functionality of AD_{kwik}.

The AdImportExportPresentationComponent is a component located in the presentation layer which offers two user interfaces, one for import and one for export. In the domain layer, the AdImportExportDomainComponent is responsible for the import and export logic. It offers interfaces with methods to import and export decision trees. To perform the import, it uses the ADkwikDomainLayer component, which is responsible for the correct updating and versioning of the decision content. The ADkwikDomainLayer accesses the AdDataSourceComponent which is located in the data source layer and performs the low-level data access on the StorageDataSourceComponent, i.e. the create, read, update, delete and search functionality (CRUDS). The AdDataSourceComponent uses an ApplicationServer as runtime, which again runs on an HttpServer.

The component models realizing other functionalities, e.g. search or dependency management, look similar, hence, they are not shown here.

To meet requirement TR350, the search functionality, the QEDWiki search mechanism can be used. This mechanism enables searches in whole wiki pages which return lists of pages containing the search term. Based on the AD attributes an advanced search mechanism can be implemented and provide a more sophisticated user interface. This allows users for instance to search for ADs with a particular decision driver or to restrict the search to a particular project.

TR360 requires the generation of documents from the AD descriptions. To fulfill this requirement a function is designed to parse the respective attributes of ADs, alternatives and outcomes into a HTML template (e.g. for ARC 100 documents).

4.4 Decision Workflow

TR400 requires the implementation of the decision making process depicted in Figure 1.1 on page 6. This decision making process applies to one AD. This means, starting a decision making process creates an outcome with a life cycle. During the architecture design process, the decision making process is executed several times, at least once for each outcome. Several decision making processes can be executed in parallel. Furthermore, in real world, following the decision making process step by step most often is not possible. The three phases can overlap or even parallel. Often iterations become necessary, or reconsiderations of ADs or alternatives force a jump back into earlier phases.

As decision making processes are conducted in parallel, several influences on outcomes amongst the decision making processes occur. These influences are captured as outcome and time dependencies through the dependency management explained in Section 4.2.2. The influences between the different phases in one decision making process are explained in the following.

The decision identification phase can be conducted simultaneously to the decision making and enforcement phases. Problem and alternative documentation are performed constantly throughout the design process. Every now and then, new alternatives are added to the pool of possible solutions, e.g. when vendors release new products or a new open source technology becomes available. Ajax, for instance, recently opens new alternatives for the AD 'Presentation Layer Technology'. If a new alternative is captured in the identification phase it also influences the outcome from the decision making phase. This can even lead to the need for changing the outcome to another alternative. Which alternative was chosen during the decision making phase, i.e. the outcome of the decision, influences the enforcement. Different alternatives can be enforced in different ways. However, to enforce ADs, e.g. through direct mapping of ADs to code, is out of scope for this thesis.

State management is the underlying principle in AD_{kwik} to capture the influences in the decision making process. The decision making process influences the state of an outcome during execution. Figure 4.10 shows the different states of the outcome. This state diagram results from interviews with the target audience, the architects, and studying recent research work about decision states like Kruchten et al. [4].

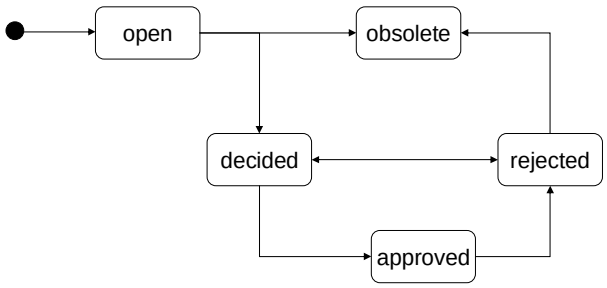


Figure 4.10: Possible states in the life cycle of an outcome.

- The creation of an AD is performed through its identification and capturing. At the same time, one or more outcomes can be created, which are in state *open*. In this phase also some alternatives are captured. To give an example, the architects from the scenario in

Section 3.1.2 create the new AD ‘Presentation Layer Technology’ and add the alternatives ‘Rich Client’, ‘Thin Client’ and ‘Rich Internet Application/Web 2.0’. The outcome of AD ‘Presentation Layer Technology’ is in state open.

- Making a decision in the second phase of the decision making process depicted in Figure 1.1 transfers the outcome into state *decided*. The architects of company XYZ decide for ‘Rich Client’ since this enables a rich user interface. The outcome of AD ‘Presentation Layer Technology’ is now in state decided.
- The outcome then can be *approved* by a responsible architect. Peter, the project lead, revises the outcome of AD ‘Presentation Layer Technology’. Being aware of the possibilities of RIAs, he wants his architects to change the decision for the ‘Rich Client’ to ‘Rich Internet Application/Web 2.0’. This means, the outcome of AD ‘Presentation Layer Technology’ is not transfered into state approved, but needs to be revised.
- If the outcome has to be changed, it first has to be *rejected* and then can be decided again. Therefore, Peter transfers the outcome into state rejected. The responsible architect of his team now chooses alternative ‘Rich Internet Application/Web 2.0’. The outcome of AD ‘Presentation Layer Technology’ is again in state decided. Now, Peter can approve it. An approved outcome can be rejected as well. If for instance, a new technology becomes available as alternative after approval of a decision, it might become necessary to reject the approved decision and change to the new alternative.
- An AD can only be deleted if all its outcomes are in state *obsolete*, which only can be reached by outcomes which are not decided or approved. If an outcome is in one of these two states, the AD is important for the architecture and therefore must not be deleted. The AD ‘Presentation Layer Technology’ for instance can not be deleted, because its outcome already is approved by Peter.

The life cycle of an outcome does not have an end, because even with the end of a decision making process, the outcome persists in the state in which the decision making process stopped. The life cycle can be restricted through the AD attribute ‘phase’ in the domain model which specifies the life span of an AD in the project. For instance, the phase attribute of AD ‘Presentation Layer Technology’ is ‘solution outline’. This means, the life cycle of its outcome happens in the solution outline phase of the project. After this phase, the outcome should not be changed.

Parallel identification and capturing of different ADs can be achieved in AD_{kwik} with the content repository and wiki structure. Moreover, the AD_{kwik} user interface offers the possibility to change the state of an outcome. These state changes are captured and versioned in the database which provides a history of state changes.

4.5 Collaboration Features

Section 2.3 introduces several Web-based collaboration tools, e.g. message board and email, which are of use for AD_{kwik}. QEDWiki itself already is a collaboration tool which includes other tools like a simple message board (the “comment” mechanism) and email for communication. Further-

more, it provides knowledge management through the wiki functionality and the possibility to upload files. Message boards allow organization of discussed knowledge through the user. Emails are automatically assigned to one QEDWiki page, thus they are attached to the AD located on this page. Furthermore, all emails are stored and accessible centrally in AD_{kwik}.

If the necessity for more collaboration features arises, the architects even could mash up the pages and build situational applications. This could be the incorporation of blogs, feeds, project plans or calendars.

For inter-project collaboration, AD_{kwik} introduces the concept of *projects* into the wiki. A project team can store and organize its data in its own projects. In these projects, decision trees and other wiki pages of the project team, like addresses of team members or information about the team, can be included. This allows the usage of the wiki by several project teams at the same time. One could even think of teams inspecting and comparing decision content of other teams.

4.6 User Interface

This section introduces the user interface concept for AD_{kwik} which supports the functionalities examined in previous sections. First of all the overall site layout is introduced. The following subsection presents the information architecture, which is base for the navigation in AD_{kwik}. Then the representation of the domain model is shown as well as the user interface for dependency management.

4.6.1 Site Layout

The QEDWiki user interface is a mixture between traditional Web site layout and rich client application design. The overall style of the application has the typical partitioning of Web applications into header, left side navigation and content area, whereas the menu resembles the drop down menus of conventional desktop applications. AD_{kwik} preserves this style because it will be useful, especially since navigation through content, in this case the decision tree, and modification of the content are both functionalities of the application.

The screen is organized into four parts: logo, navigation, content explorer and content area. Figure 4.11 shows these four parts of the screen. The menu in the navigation is split into two parts and contains the operations to create new content or modify content as well as switch between different perspectives. The content explorer contains the decision trees, which can be included in projects. When selecting one entry in the tree in the content explorer, the relevant content will be displayed in the content area. This concept is also known as Master-Details-Pattern [45] (pages 138ff.). The pattern consists of two parts: the master section and the details section. The master section usually consists of a list or tree structure, whereas the details section displays content depending on the selection in the master. Examples of this patterns can be seen in application user interfaces like the one of TeXnicCenter (LaTex editor), Adobe Reader when used with bookmarks or the Windows explorer.

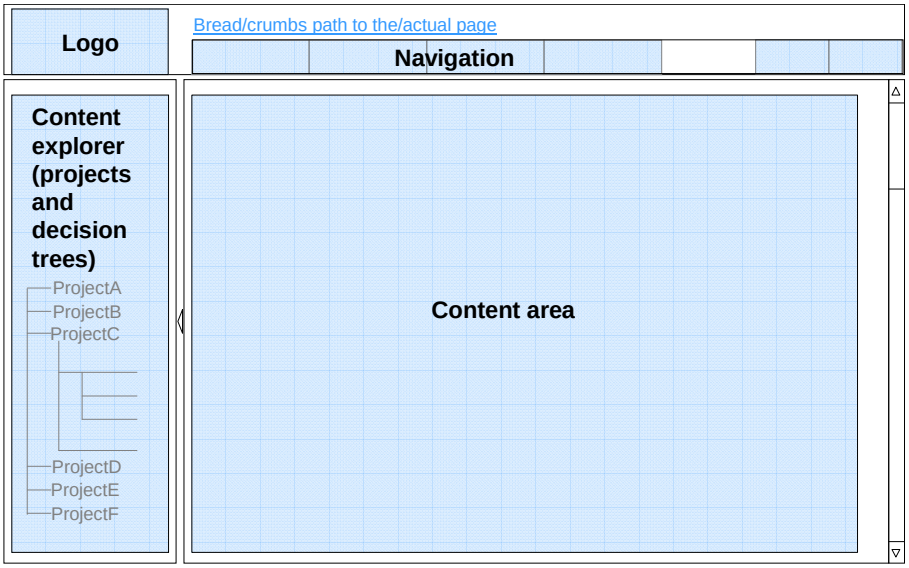


Figure 4.11: The layout of the AD_{kwik} user interface using the Master-Details-Pattern [45] (pages 138ff.).

The key point of the Master-Details-Pattern is, that the master is always visible to the user. This results in the benefit, that the details can be reached quickly since they are represented through a node in the master. Through the master view, the user furthermore can get and keep an overview of the whole content. Using the Master-Details-Pattern addresses two important aspects of AD_{kwik}: fast information access and keeping an overview of the knowledge all the time.

4.6.2 Information Architecture and Navigation

The information architecture of the application represents the use cases examined in Section 3.1.3 and the wiki functionality which is already implemented in QEDWiki. Information architecture is a term from the user interface design domain, denoting the design of "organization, labeling, navigation, and searching systems" [48]. This includes grouping and naming content and establishing navigation and search mechanisms to support the user to find and manage information easier and faster. The information architecture for AD_{kwik} is shown in Figure 4.12.

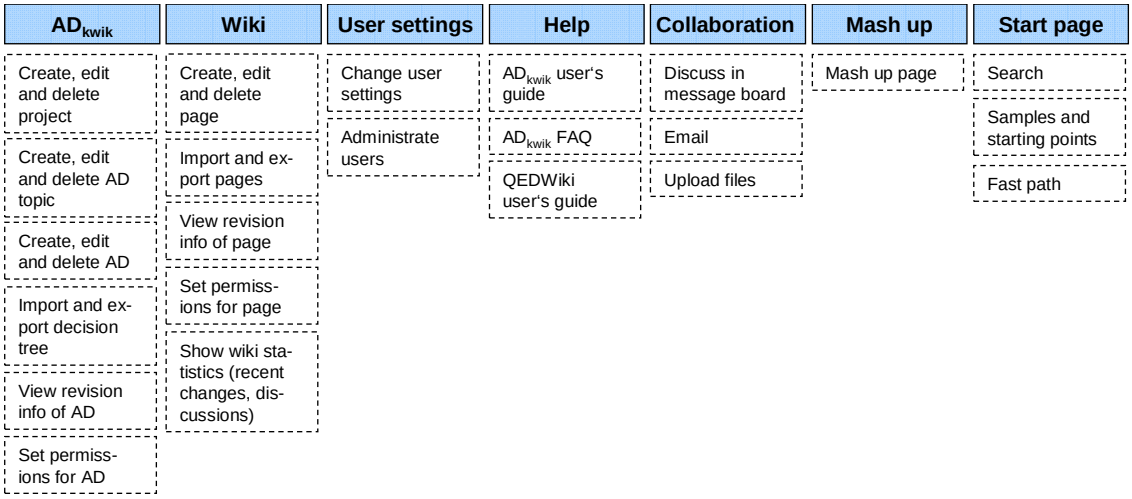


Figure 4.12: Information architecture of AD_{kwik}. Categories are located in the top row.

The information is classified into different categories. Categories *Wiki*, *User settings*, *Collaboration* and *Mash up* are provided by the QEDWiki and are used as they are. Category *AD$_{kwik}$* includes several new functions to create, edit, delete and import AD$_{kwik}$ specific content. Category *Help* is also provided by the QEDWiki but is enriched by the AD$_{kwik}$ user's guide and a list of frequently asked questions (FAQ). The category *Start page* represents the entry point into the application. It contains search functionality and AD$_{kwik}$ starting points. Guidance for different types of users should be included into the Start page. This could include roadmaps like sample scenarios for the novice users or the most important functionalities for fast access. The *fast path* concept provides example walk-throughs for different types of projects. Small projects often do not need to inspect the whole AD content. A small set of relevant ADs provided at project start give an overview about the ADs to make and an entry point into the decision making process. To give an example, Peter and his team from the scenario explained in Section 3.1.2 design and develop a typical small SOA project without need to implement process logic. With the fast path concept, AD$_{kwik}$ provides them a small set of ADs which are relevant for a simple SOA project. This helps with getting started and provides guidance through the most important ADs.

Providing two distinct categories for AD$_{kwik}$ and plain wiki provides the ability to use either of those functions. The management of wiki pages is logically separated from the management of ADs which includes dependency management and decision workflow. This results in a smaller number of menu items for each category. Fewer entries help the user to better orient in the menu and to faster learn the menu structure.

The structure in the navigation menu of AD$_{kwik}$ is based on this information architecture. QEDWiki provides the possibility to switch between displaying pages, discussing content and mashing up pages. This feature is located on the right hand side of the navigation menu. User tests showed that this location is easy to reach and the features are intuitive to use. Therefore this strategy is kept in AD$_{kwik}$. Besides the menu the user navigates through the content via the tree structure in the content explorer. The tree implements a mechanism to collapse and expand, since the decision tree can get very large.

Another navigation feature is implemented in AD$_{kwik}$: The path to the actual page, which is located above the navigation menu. Figure 4.11 shows an example for it above the navigation part. This path implements the breadcrumbs usability pattern [49], which helps the user with the navigation through a hierarchy of pages in that it maps the hierarchy to a flat path.

4.6.3 Representation of the AD Content

The content of each AD, that means the AD attributes, alternatives, dependencies and the outcome are displayed in the content area of the front-end. AD and alternatives fit on one screen with a resolution of 1024x768 or more. This means, they are visible without scrolling. Information can be inspected at one glance which results in faster access and better usability. To ensure this feature, the length of the attribute values is limited to 2000 characters per attribute. Sample context from the SOA Community of Practice showed, that this length is large enough to express the attributes.

Figure 4.13 shows the organization of the AD attributes in two columns. The division of the content into several columns ensures to use as much of the visible space as possible without confusing the user. The AD attributes are furthermore divided into several boxes which group them together logically. The alternatives are included directly into the AD to avoid unnecessary navigation steps between different alternatives and the AD. The user can inspect all alternatives without leaving the AD which they belong to. The outcome is included at the bottom most box, including a button for deciding.

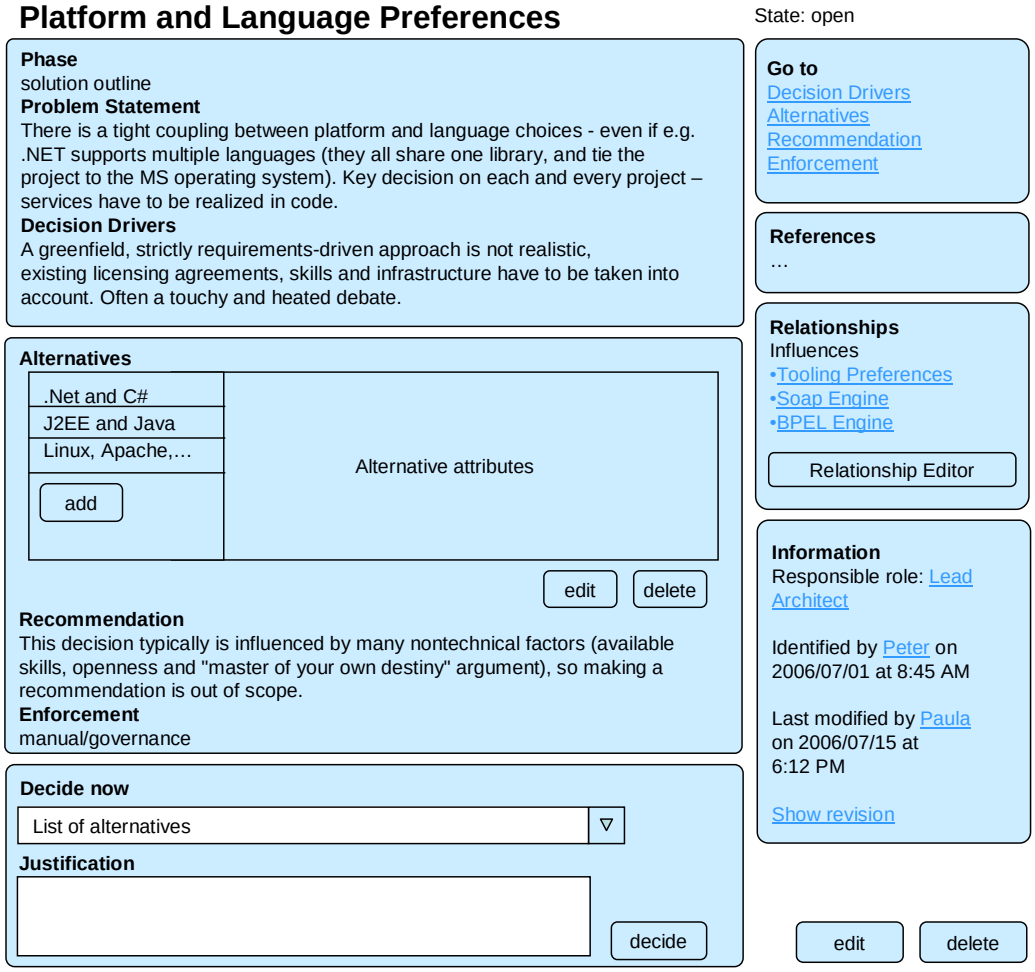


Figure 4.13: Structuring of the AD content in AD_{kwik}.

To edit or delete the AD, buttons are provided on the bottom of the page. The edit screen of an AD looks similar to the display screen, hence is not shown here. It contains white edit fields for each AD attribute as well as save and cancel buttons.

The decision workflow is implemented in the user interface as follows. States are depicted in the upper right corner. State changes are caused through actions. This means, if an architect chooses an alternative, the state transfers into ‘decided’. On the outcome display page, buttons allow the architect to approve or reject the decision. Which user role is responsible for the decision is displayed in the information box in the lower right corner. Only users with this role can select an outcome. In the information box, also the concrete person who identified and the last person who modified the AD are shown.

Dependency visualization is implemented in the right column in Figure 4.13. The relationships box shows how the dependencies can be shown as hyperlinks. Furthermore, a button is provided, which leads to the relationship editor. The relationship editor provides an interface to delete existing and create new dependencies between ADs and their alternatives. Figure 4.14 shows the user interface to edit the dependencies. In this case, the selected alternative 'Linux, Apache, MySQL, PHP (LAMP)' of AD 'Platform and Language Preferences' is the originator of the dependency. It forbids the alternative 'Java POJO' of AD 'Provider Type'. Therefore the influence type 'forbids' is chosen.

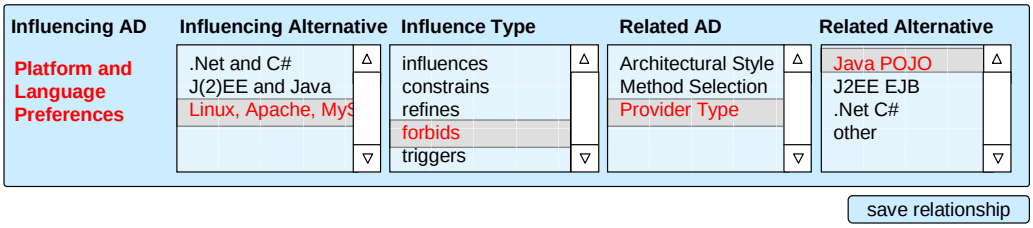


Figure 4.14: Relationship editor.

The conceptual design of AD_{kwik} described in this chapter is implemented in a prototype. The next chapter highlights details of this implementation.

5 Implementation of AD_{kwik}

The previous chapter describes the conceptual design of AD_{kwik}. This chapter outlines how these concepts are implemented in QEDWiki, PHP and the Zend framework [39]. The explanation is not meant to be exhaustive, but gives an overview about how the layers of AD_{kwik}, i.e. data source, domain and presentation, are implemented and look like.

5.1 Model and Persistence

5.1.1 Data Model

The domain model proposed in Section 3.1.4 is mapped to five tables in the QEDWiki MySQL database. Figure 5.1 shows a simplified entity-relationship diagram which depicts the tables as entities. Each table includes a unique identifier which is specified in the *id* field. Since ids build the primary keys of the tables, Figure 5.1 depicts them in bold.

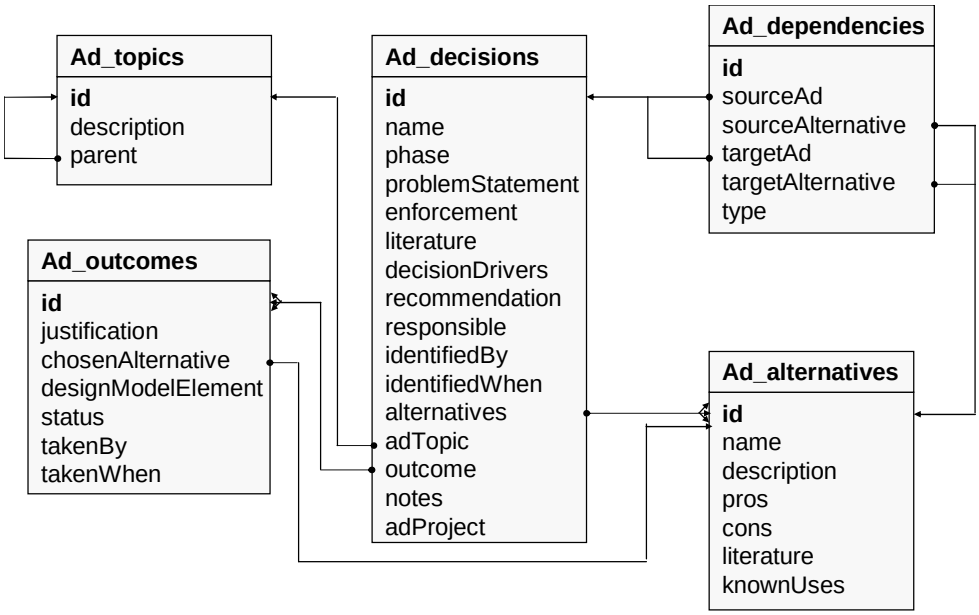


Figure 5.1: Data model of AD_{kwik}.

- *Ad_topics*: Contains description elements of the domain model classes AdTopic and AdLevel. The *parent* field links the id of the parent topic or level. If this field is empty, the AD topic represents an AD level. AdTopic and AdLevel are stored in the same table, since their attributes, thus the columns in the table, are the same. An AD level is an AD topic without parent.

- *Ad_decisions*: Contains the elements of the domain model class AD. The association to the including topic is realized with the field *adTopic* which directs to the id of the AD topic in the table Ad_topics. The *alternatives* field can contain a list of several ids of instances of the Ad_alternative table. The *outcome* field can link to ids of Ad_outcomes instances. This is depicted through the multiple arrows at the association end.
- *Ad_alternatives*: Contains the elements of the domain model class AdAlternative.
- *Ad_outcomes*: Contains the elements of the domain model class AdOutcome. The field *chosenAlternative* links the id of an instance of Ad_alternatives.
- *Ad_dependencies*: Contains fields for the capturing of dependencies. This includes the *sourceAd* and *sourceAlternative* as well as the *targetAd* and *targetAlternative*, which contain the ids of the respective instance in tables Ad_decisions and Ad_alternatives.

The mapping of the respective objects in the memory to these tables is performed by Active Records. How it is implemented in AD_{kwik} is explained in the following.

5.1.2 Active Record Pattern

The object-relational mapping of wiki objects to the tables described above is done with the help of the Active Record pattern. This is proposed in Section 4.3. Analogous to the implementation of the QEDWiki wiki objects, the implementation of the Active Record pattern in AD_{kwik} is done by creating one class for each AD_{kwik} table which extends the ZActiveRecord class. Example source code in PHP, the Active Record class *Ad_alternative*, is shown in Listing 5.1.

Listing 5.1: Active Record of alternative.

```
1 /**
2 * @package   AdEntities
3 */
4
5 class Ad_alternative extends ZActiveRecord{
6
7 }
```

The Zend Framework automatically relates this class to the table Ad_alternatives, because the names of table and class are the same. Variables are assigned to the class, which correspond to the fields of the table. Furthermore, the class implements create, read, update and delete (CRUD) methods.

Listing 5.2 shows that the usage of the Active Record is straightforward and only needs little coding effort. In line 2 the creation of a new record is shown. The name and the description fields are then assigned with values. By calling the save()-method in line 5 the Zend Framework creates a new database record with a new index in the table Ad_alternatives, and stores the attribute values into this database record.

Listing 5.2: Creating, storing and searching for an Active Record.

```
1 /* create a new alternative with name and description */
2 $alternative = new Ad_alternative();
3 $alternative->name = "Thin client";
```

```
4 $alternative->description = "...";
5 $alternative->save();
6
7 /*search alternative with name */
8 $a = new Ad_alternative();
9 $search = array (
10         name => "Thin client", //name of the alternative
11 );
12 $alternative = $a->findFirst($search);
```

Moreover, the Active Record provides functions to search the table via arguments. As argument for the findFirst()-method in the example, an array is used. The findFirst()-method turns the array into a SQL query and returns the first record found which matches this query.

Each Active Record save and update operation is done in one transaction using autocommit.

The following outlines, how the Active Records are versioned and how AD content import is implemented on the domain layer of AD_{kwik}.

5.2 Domain Logic Implementation

5.2.1 Versioning

As described in Section 4.3 the Edition pattern is used in order to version the records in the database. For each AD_{kwik} table shown in Section 5.1.1, versioning is implemented.

The class *AdAssetManager* serves as originator for the editions and is responsible for managing the AD_{kwik} wiki objects. This class implements methods to create and update the objects with versioning, which means an extension to the Active Record functions. An edition is represented by an Active Record instance in the memory which in turn is represented by one record in the database. The fields of this database record build the memento. To assign the event to the edition, each table has three additional fields:

- *modifiedBy*: User name of person who modified the record.
- *modifiedWhen*: Timestamp of the moment the new record was saved.
- *isLatest*: Flag, if the edition is the newest of all editions. Needed for faster access on the newest record in the database.

Listing 5.3 shows the source code to update an alternative. The parameters of the function are the id of the Ad_alternative which has to be updated and the data which has to be assigned to the new edition of the alternative. In line 4, the actual edition of the Ad_alternative is collected. The flag of this old version is set to 0, which marks it as outdated. It is then saved via the Active Record save()-method. In line 10 a new Ad_alternative is created and the new data is assigned to this new version. The isLatest flag is set to 1, which marks this edition as the newest. Finally, the newest version is saved and returned.

Listing 5.3: Updating an alternative in the AdAssetManager.

```
class AdAssetManager{
   //...
   function updateAlternative($alternativeId, $alternativeData)
       {
        $oldAlternative = getActualAlternative($alternativeId);
        // Set the isLatest flag to no and save the old
           alternative.
        $oldAlternative->isLatest = 0;
        $oldAlternative->save();

        // Create a new Ad_alternative and assign the new data
           to it.
        $newAlternative = new Ad_alternative();
        $newAlternative->getAttributes($alternativeData);
        //old and new AD have the same ID
        $newAlternative->id = $alternativeId;
        // Set the isLatest flag to yes and save the new
           alternative.
        $newAlternative->isLatest = 1;
        $newAlternative->save();
        return $newAlternative ;
   }

   function getActualAlternative($altId) {
        $search = array (
                id => "$altId",
                isLatest => "1"
        );
        $alternative = new Ad_alternative();
        $alternative = $alternative->findFirst($search);
        return $alternative;
   }
  //...
}
```

5.2.2 Content Import

For the pre-population of the decision tree, an import script for the Architects' Workbench (AWB) is implemented. The AWB manages information about the design and delivery of IT-architecture and also implements the possibility to create and manage architectural decisions. The AWB uses its own domain model for the decisions. The instances of the domain model are stored in XML files.

The realization of the AWB import requires a mapping of the AWB domain model elements to the AD_{kwik} domain model. The AWB domain model contains AD, relationship and topic equivalents, which are mapped to the AD_{kwik} elements AD, dependency and AD topic, respectively.

As explained in Section 4.3, the implementation of the import is distributed over all three architecture layers, the presentation, domain and data source. On the domain layer the *AwbImporter* class realizes the IAdImport interface depicted in Figure 4.9. It uses an event-based XML parser which is part of the PHP core for parsing the XML files. Whenever the parser hits a start or an end tag an event is triggered. The third event, besides start and stop, is the parsing of character data in between two tags. The parser then calls the defined callback functions.

5.3 User Interface

This section explains how the AD_{kwik} presentation layer components are realized with the help of QEDWiki and how the QEDWiki user interface is adapted to the needs of AD_{kwik}. Furthermore, it presents selected screenshots of the prototype in order to explain the realization of dependency management and decision workflow.

5.3.1 Layout Overview

As proposed in Section 4.6.1, the user interface of the prototype uses the QEDWiki structuring into content explorer, navigation menu and content area. The only changes made to the QEDWiki layout are changes in the information architecture of the navigation menu. The new menu added to the menu bar is depicted in Figure 5.2, highlighted by the red oval on the left side. This menu contains the menu items which are proposed in Section 4.6.2. Another change of the QEDWiki layout is the renaming of the "Assemble" function into "Mash up" and "Share" into "Collaborate" which is depicted in the green oval on the right. User tests on the old terminology showed, that these terms were not intuitive, thus need to be changed.

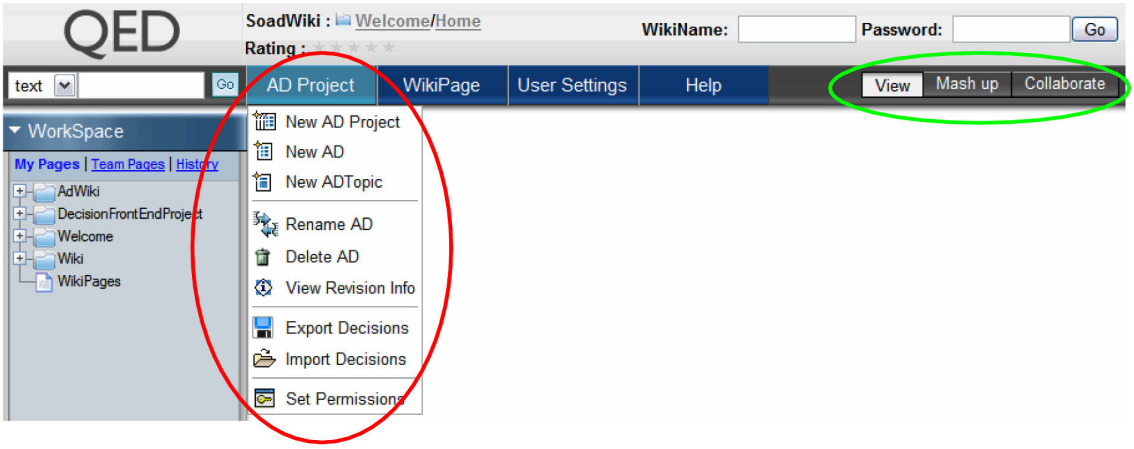


Figure 5.2: Layout of AD_{kwik} and navigation menu.

Pages in AD_{kwik} are displayed in the content area and can contain ADs, AD topics or AD projects. The implementation of this extension to QEDWiki is explained in the following.

5.3.2 Implementation of User Interface

The representation of the AD_{kwik} content, i.e. ADs, AD topics and AD projects, is implemented using the command/widget mechanism of QEDWiki which is explained in Section 3.2.3. At the time of writing there are four commands: *AdCommand*, *AdTopicCommand*, *AdProjectCommand* and *AdImporterCommand*. Each of these commands implements presentation logic, i.e. the processing of content for displaying in the user interface. The AdCommand implements a user interface to create, edit and delete ADs as well as alternatives, dependencies and outcomes. The AdTopicCommand lets the user create and edit AD topics. The AdProjectCommand is used to create AD

projects. The AdImporterCommand provides a user interface for the import of AD content from other sources, e.g. the AWB.

Listing 5.4 shows an extract of the AdTopicCommand. This listing demonstrates how commands are implemented in QEDWiki.

- Each command extends the QEDWiki class WikiCommandWithMetadata (line 1).
- A command can be parametrized in the constructor (lines 3-8). Values for these parameters can be assigned in the widgets in the user interface.
- When a page is called, which includes the command, the constructor and the renderBody()-method are called (line 13). Therefore user interface and presentation logic is implemented in this function.
- External classes like the AdAssetManager can be used (line 15), thus logic processing can be sourced out into external classes and be reused.
- A command can contain any HTML code which is then shown on the wiki page (lines 56-60). This allows individual formatting of AD content.
- As explained in Section 3.2.3, the command code is directly embedded into the called page at rendering time. Therefore, it can use the HTTP GET or POST methods to process forms and thus implement simple page flows. The AdTopicCommand simply checks which data was sent by the user and acts accordingly.

Listing 5.4: Extract of the AdTopicCommand.php.

```
1  class AdTopicCommand extends WikiCommandWithMetadata {
2    /** Constructor: Assignment of the specific ids. */
3    public function __construct($id = null, $parentNode = null) {
4      parent :: __construct($id, $parentNode);
5      $this->addParameter('id','id','N','id of command');
6      $this->addParameter('topicId','topicId','N','id of topic');
7      $this->addParameter('adProject','adProject','N','project
           name');
8    }
9    private $pageMgr;
10   private $adMgr;
11
12   /** Render HTML content. */
13   public function renderBody() {
14     $this->pageMgr = new AdPageManager(); //responsible for
           updating presentation layer
15     $this->adMgr = new AdAssetManager(getDbHandleFromQEDWiki());
            //responsible for data souce/domain layer
16     //check the posted data in $_POST
17     //if save button was pressed
18     if ($_POST['save'] == 'save') {
19         //update database with new data and display the AD topic
20         $topic = $this->adMgr->getActualAdTopic($this->
               getArgument('topicId'), $this->getArgument('adProject
               '));
21         $topic->description = $_POST['topicdescription'];
22         $topic->modifiedBy = getTheUserIdFromQEDWiki();
23         $topic->modifiedWhen = date('YmdHis');//date in a
               special format for storing in DB
24         $this->adMgr->updateAdTopic($topic);
25         $this->showTopic();
26     }
```

```
27      //if edit button was pressed
28      elseif ($_POST['edit'] == 'edit') {
29          //display edit form
30        $this->showEditTopicForm();
31      }
32      //if save button was pressed on the create AD topic page
33      elseif ($_POST['savenew'] == 'save') {
34        //create new db record for topic
35        //...
36        //create new wikipage for AD topic using the AdPageManager
37        $this->pageMgr->createTopicPage($newtopic,$_POST[parent],
              $newtopic->adProject);
38        redirectToPage($_POST[parent] . "/" . $_POST['topicname'])
              ;
39      }
40      //if cancel button on the create AD topic page was pressed
41      elseif ($_POST['cancelnew'] == 'cancel') {
42          //redirect to the previous page
43        redirectToPage($this->pageMgr->getPreviousPage());
44      }
45      else {
46        $this->showCreateTopicForm();
47      }
48    }
49    /** display the edit form for the AD topic */
50    private function showEditTopicForm(){
51      //...
52    }
53    /** display the AD topic */
54    private function showTopic(){
55      $topic = $this->adMgr->getActualAdTopic($this->getArgument('
            topicId'), $this->getArgument('adProject'));
56      echo "<h1>" . $topic->name . "</h1><br>";
57      echo "<form method='post'>" .
58             "<div class='decisionbox' >" . $topic->description .
                   "</div>" .
59             "<input type='submit' name='edit' value='edit'>.
60           "</form>";
61    }
62  }
```

The other commands are implemented analogously.

After this explanation of presentation logic implementation details, the following shows how the user interface looks like.

5.3.3 Selected User Interface Screens

The AdProjectCommand implements the project concept introduced in Section 4.5. The user can create virtual projects by selecting the 'Create AD Project' menu item in the AD Project menu explained in Section 5.3.1. AD_{kwik} displays a form in which the user can input the name of the project. The project team from the scenario described in Section 3.1.2 for instance creates a project with the name "DecisionFrontEndProject". After pressing the save button, the screen depicted in Figure 5.3 shows up. In the page tree on the left hand side, a new page for the project appears on the root level. After the creation of a project, different actions can be performed. A selection of useful steps is shown in the content area and linked to the respective action.

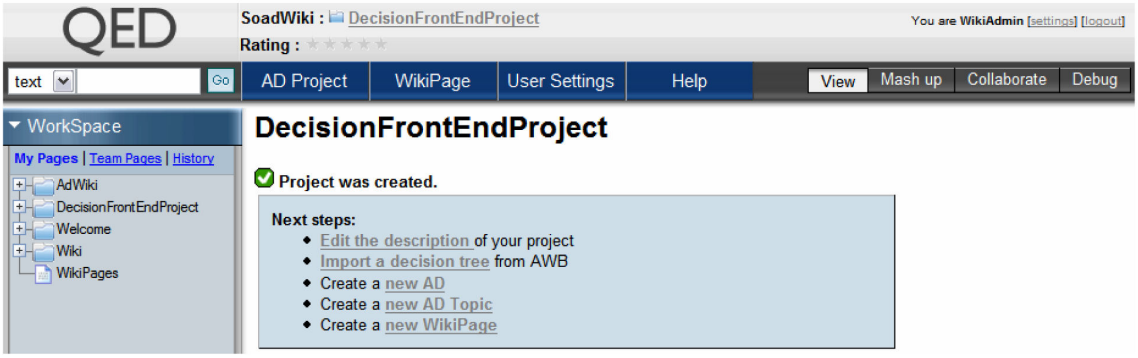


Figure 5.3: Screen after creation of an AD project.

This concept of next steps is applied at several places in AD_{kwik} in response to the usability requirement outlined in Section 3.1.5. It is used as instrument to guide the user through the different processes in the wiki.

AD with Embedded Alternatives

One important AD for the decision front-end project is 'Presentation Layer Technology', which is shown in Figure 5.4. The screen is realized as proposed in Section 4.6.3. The alternatives are embedded into the AD description. The view of the alternative is split into two parts: The list of the alternatives on the left and the description of the selected alternative on the right. Furthermore several buttons to add, delete, and edit alternatives are provided. The selected alternative in Figure 5.4 is 'Rich Client', therefore the description of this alternative is displayed. The presentation of the alternatives has the color light gray whereas the rest of the AD description is light blue to ensure better readability.

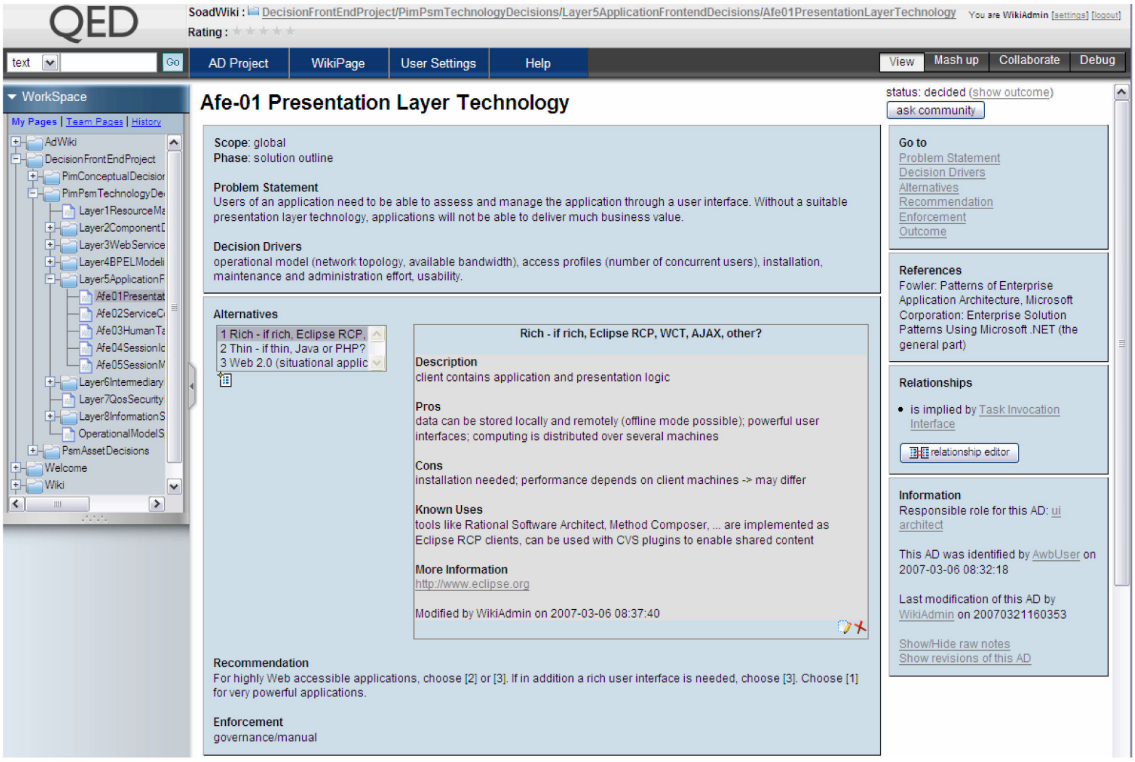


Figure 5.4: Display of alternatives embedded into the AD description.

If the user selects an alternative on the left, the respective alternative description is shown dynamically without reloading the whole page. This is implemented with the help of the Dojo toolkit [50]. The selection of an alternative on the left calls a JavaScript function parametrized with the attribute values of the respective alternative. The JavaScript function on the client side loads a HTML template which is located on the server and fills it with the attribute values. It then updates the part of the page on the right with the loaded and filled HTML template.

The relationships box on the right hand side of Figure 5.4 contains a button to the relationship editor. The following explains the functionality behind this button.

Dependency Management

The relationship editor button on the right hand side of an AD page directs to the relationship editor, which is depicted in Figure 5.5. As outlined in Section 4.2.2, a dependency can exist between two ADs and their alternatives and can have different influence types. The figure shows one dependency for the AD 'Presentation Layer Technology': AD 'Presentation Layer Technology', the source AD, is influenced by AD 'Task Invocation Interface', the target AD. The relationship field shows the influence type for this dependency. The user can add new dependencies with the help of the editor on the bottom of the page. One architect of the decision front-end project discovers that the dependency between AD 'Presentation Layer Technology' and AD 'Task Invocation Interface' has a stronger influence type than 'influences'. He therefore selects 'refines' as influence type between the two ADs which is depicted in the figure. After storing the new dependency, he can delete the old one with the weak 'influences' dependency type. With the relationship editor architects can manage outcome dependencies as well as time dependencies as influence type 'triggers' can be selected.

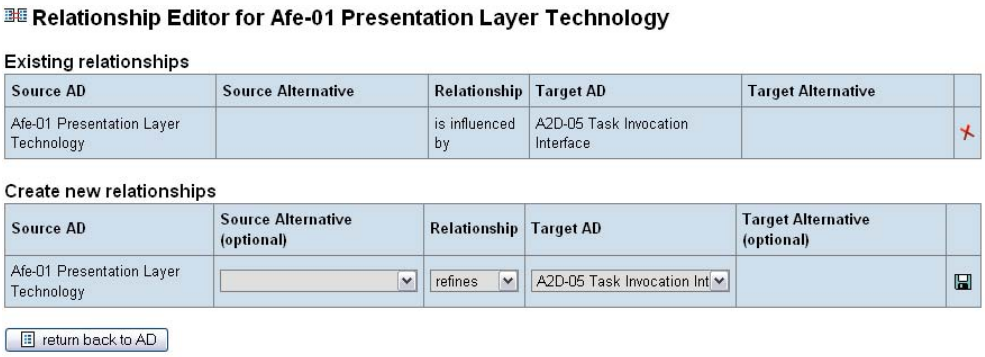


Figure 5.5: Relationship editor to manage several kinds of dependencies.

Decision Workflow

As explained in Section 4.4, the decision workflow is realized in states of outcomes which can be changed through interaction of the user. The state change from open to decided is triggered by creating an outcome for an AD. Figure 5.6 shows the form which provides this functionality. For instance, the responsible architect in the decision front-end project selected the alternative 'Rich Internet Application' and can now decide for this alternative by pressing the decide button.

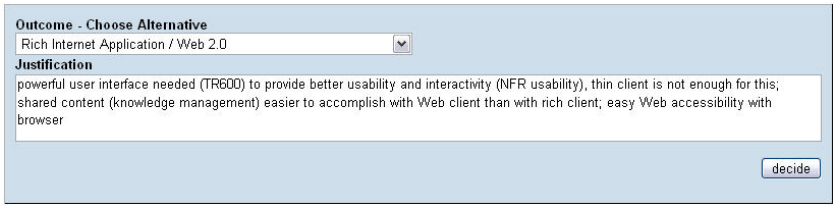


Figure 5.6: Form to create an outcome, i.e. decide for an alternative.

After pressing the decide button, the page is reloaded and the outcome is displayed as shown in Figure 5.7. The description of the outcome includes the chosen alternative, the justification for it and why not choosing the other alternatives. Furthermore, information is displayed, who decided for this alternative at which point of time.

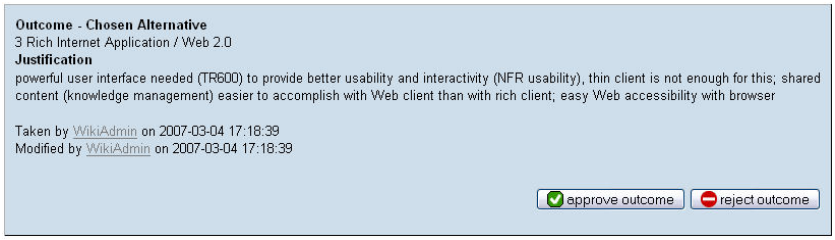


Figure 5.7: AD was decided by WikiAdmin.

If the decision was not taken for the first time, the modify date differs from the taken when date. With the provided approve and reject buttons, a responsible architect, e.g. the project lead architect, can now approve the outcome or reject it which transfers it into the respective state. After rejection, the screen in Figure 5.6 is displayed again. However, the database now contains the information, that the decision has been rejected and why.

The actual state of a decision is shown on the top row of the AD description. The decision from the previous figures is in state ‘decided’, which is depicted in Figure 5.8.

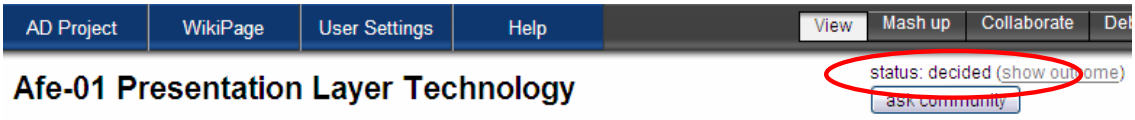


Figure 5.8: Status display: AD is in state ‘decided’.

Collaboration Features

As suggested in Section 4.5 the collaboration features of QEDWiki are used. This includes a message board, email and a file upload mechanism. Each page has its own collaboration page. The collaboration features can be accessed by switching the perspective from View to Collaborate in the right hand navigation menu. Figure 5.9 highlights this in the red oval and shows the message board feature in the content area of the screen. The architects of the decision front-end project use the message board to discuss about adding the alternative ‘Rich Internet Application’ to the AD ‘Presentation Layer Technology’.

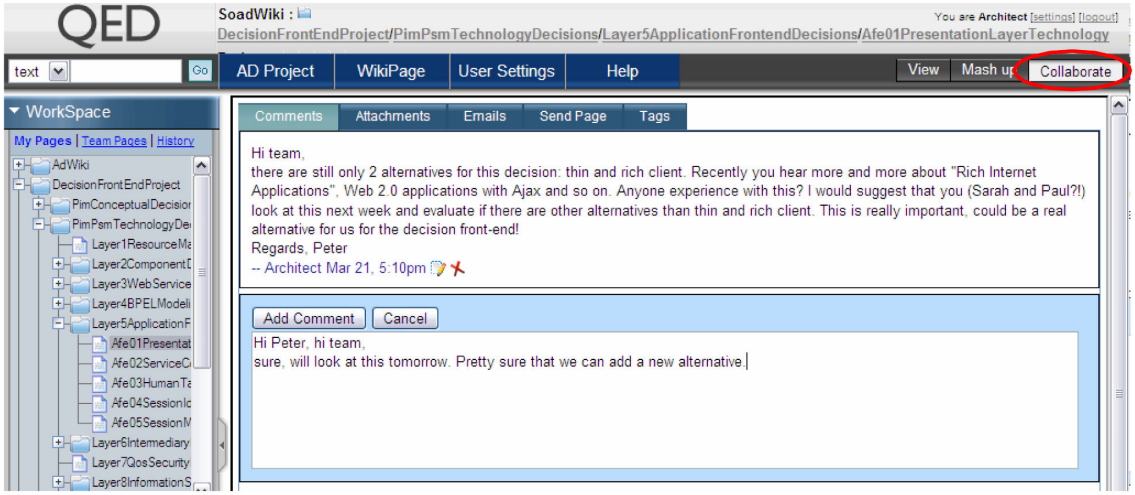


Figure 5.9: The message board feature of QEDWiki in the Collaborate perspective.

5.4 Code Statistics and Used Tools

The implementation of AD_{kwik} comprises around 4800 lines of code, including 1500 lines for comments. QEDWiki code is not included in these numbers. The source code includes PHP, JavaScript, HTML and CSS. There are 16 PHP classes, 4 of which are plug-ins to the QEDWiki.

For the implementation the following tools were used:

- Eclipse IDE 3.2.1 [36]
- Apachefriends XAMPP, Version 1.5.3a (basic package) [51]
- QEDWiki with Zend Framework and Dojo Toolkit, Version 1.0.0, Build 100906 [37]
- PHPeclipse plug-in, Version 1.1.8 [52]
- Browsers: Microsoft Internet Explorer, version 6 and Mozilla Firefox, version 1.5
- Firefox JavaScript Console: Debugging JavaScripts
- Rational Software Architect, Version: 6.0.1: Documentation of design [53]

Whether the concept described in Chapter 4 and the prototype implementation of AD_{kwik} in this chapter meet the requirements, how they differentiate from related work and if they are accepted by the first users is described in the next chapter.

6 Evaluation of AD_{kwik}

The previous chapters outline the concept and implementation of AD_{kwik}, a system for AD modeling, making and knowledge management. This chapter evaluates AD_{kwik} against the requirements gathered in Chapter 3.1. A comparison to related work then elaborates the unique differentiators of AD_{kwik}. Finally, first user feedback reveals, whether AD_{kwik} promises to be accepted and used by the architects community.

6.1 Evaluation of AD_{kwik} against Requirements

As the use cases captured in Section 3.1.3 lead to the requirements described in Section 3.1.4, this section evaluates AD_{kwik} against the functional and non-functional requirements. The use cases are consulted in order to evaluate the user interface.

TR100 Domain Model for Architectural Decisions
This requirement is met by the proposed domain model in Section 4.2. The domain model provides a structuring of ADs, alternatives and dependencies as well as outcome elements. ADs are organized as nodes in the decision tree and further classified into AD levels and topics. AD_{kwik} maps the domain model to the data source layer with the data model explained in Section 5.1.1. The user interface on the presentation layer realizes functions to modify instances of the model.

TR200 Dependency Management
The required visualization of dependencies is realized through the dependency management concept in Section 4.2.2. The dependencies are captured in the database as foreign key relationships between tables. The extensibility of the dependency management is ensured through the possibility to implement domain logic on top of this data.

TR300 Content Repository
The sub-requirements of TR300 are realized as follows: The data storage is provided by the database and the file system on the Web server. Active Records realize the object-relational mapping. Versioning of the data is realized with the Edition pattern on domain layer (Section 4.3). This provides a mechanism to implement versioning in the user interface as well. The AdAssetManager provides a simple API to the data which can also be used by a Web service. An example import mechanism for AWB is implemented as explained in Section 5.2.2, an export mechanism is part of the conceptual design. Advanced search based on AD attributes as well as the generation of documentation based on HTML templates are part of the concept.

TR400 Decision Workflow

A concept for the decision process support is introduced in Section 4.4. The state management proposed in this section is implemented in AD_{kwik}. The implementation supports capturing new ADs and collecting alternatives, documenting outcomes as well as approving or rejecting decisions.

TR500 Collaboration Features

Collaboration is supported by the usage of QEDWiki features, like message board and email, as outlined in Section 5.3.3. QEDWiki already provides a user management with an authentication mechanism.

TR600 User Interface

As required, the user interface of AD_{kwik} supports inspecting and editing instances of the domain model, navigation through the decision tree and dependencies, the visualization of dependencies through hyperlinks, following the decision workflow as explained in Sections 4.4 and 5.3.3 and collaboration features. The information architecture proposed in Section 4.6.2 is implemented in the menu structure of AD_{kwik}. The evaluation of the user interface against the required use cases outlined in Section 3.1.3 showed, that all use cases are already implemented except UC700-UC900, which are search, anonymization and content export. However, these three use cases are part of the concept.

Scalability

Scalability tests showed, that the response time for page reloads stays the same if increasing the number of records in the database. The time needed for the import of decision trees increases linear to the increasing number of records (see Table 6.1). An increasing number of users performing memory-intensive operations like import increase the response times. However, importing is a rare task. As table 3.1 on page 29 shows, none-memory-intensive operations like reading and editing are much more likely.

number of AD records in database*	seconds needed for import of 130 ADs
0	34.49
130	35.94
260	37.72
2000	47.75

*plus 1-6 alternative, 0-10 dependency records for each AD

Table 6.1: Extract of scalability test.

Performance

On machines with at least 2GB RAM and a 3Ghz CPU, the response times are within few seconds: The time needed for page loads is around 1-5 seconds, when several users are online at the same time inspecting content. This is comparable to rich client applications. However, as heavy operations like import need 30 seconds and more (see Table 6.1) and decrease the performance of the whole system significantly, there is need for improvement.

Usability

The application is customized to the AD domain which enables good usability. Moreover, several usability best practices, e.g. the Master-Details-Pattern, are realized. Section 6.3 describes first user experience with AD_{kwik}.

Data Integrity

Foreign key constraints in the data model and a user interface which prevents users from inserting invalid terms sustain data integrity.

Maintainability

Due to the layering of the underlying architecture and the object-oriented approach used, AD_{kwik} meets the maintenance requirements. A developer's guide for AD_{kwik} helps with getting into the details of design and implementation.

Documentation

A user's guide for AD_{kwik} is provided in AD_{kwik} itself as wiki pages. Furthermore, a documentation for developers is delivered.

6.2 AD_{kwik} Compared to Related Work

As seen in Chapter 1, the AD domain comprises decision process, AD modeling, and collaborative AD knowledge management. This section compares AD_{kwik} to related work in the AD domain and in the broader domain of CSCW, especially knowledge management and tool support for software development teams. Figure 6.1 shows a matrix of related function blocks (columns) and recent work (rows).

	AD modeling and documentation	AD making process support	Software development support	Collaboration support	Knowledge management support	CSCW	Web 2.0 application
Ad_{kwik}	**yes**, domain model and tool support	**yes**, through dependency mgmt., decision workflow, guidance	**yes**, supports architecture design process	**yes**, wiki, email, message board	**yes**	**yes**	**yes**, application wiki
AD research projects (Kruchten et al., PAKME, Archium, etc.)	**yes**, different domain models/ontologies	**yes**, different concepts	**yes**, supports architecture design process	**(x)**, support in some projects (e.g. PAKME)	**yes**	**yes**	**(x)**, depends on implementation, PAKME is at least Web application
AD supporting software development tools (AWB)	**yes**, AWB implements ARC100 Template	**yes**, outcome element	**yes**, supports architecture design process	**no**	**yes**	**yes**	**no**, rich client (Eclipse)
Design Pattern Libraries (e.g. Yahoo! Design Pattern Library, Portland Pattern Repository)	**no**, not for ADs but for design patterns	**no**, not for ADs but lower level	**yes**, supports software development process	**(x)**, available in some of implementations	**yes**	**yes**	**(x)**, depends on implementation
Enterprise wikis (e.g. TWiki, TikiWiki)	**no**, more generic	**no**, more generic	**(x)**, depends on kind of usage/ content	**yes**	**yes**	**yes**	**yes**, all wikis are Web 2.0 applications
Decision Support Systems (e.g. Data Warehouse)	**no** domain model, no AD workflows and dependencies	**(x)**, provides information for analysis, depends on information captured	**(x)**, depends on kind of usage, often for support of management	**(x)**, available in some of implementations	**yes**	**yes**	**(x)**, depends on implementation, often rich client
Software modeling tools (e.g. Rational Software Architect, Together Architect, UML tools)	**no**, more generic, mostly focus on solution	**no**, more generic, mostly focus on solution	**yes**, support different aspects of architecture/software design/development	**no**, usually not	**(x)**, depends on kind of usage/ content	**yes**	**no**, rich clients
Content repositories (e.g. JSR-170 Java Content Repository API)	**(x)**, depends on kind of usage	**no**	**(x)**, depends on kind of usage/ content	**(x)**, available in some of implementations	**yes**	**yes**	**(x)**, some applications
Data modeling, classification, association (e.g. HyTime, Topic Maps, DocBook)	**(x)**, depends on kind of usage	**no**	**(x)**, depends on kind of usage/ content	**no**	**yes**	**yes**	**no**

(x) conditionally available

Figure 6.1: AD_{kwik} and related work.

The matrix lists projects in AD domain research, tools in the AD domain (Architects' Workbench [13]) and architecture design in general (e.g. design pattern libraries, software development tools) as well as tools for collaboration (e.g. enterprise wikis like TWiki [29], some decision support systems), more general knowledge management applications (e.g. content repositories, decision support systems) and approaches to express data and associations between data. Many

applications fall in more than one function block. This section discusses only the most relevant related work in order to reveal important differences and ideas for future work in AD_{kwik}. Thereafter, a conclusion shortly discusses relevant conceptual differences to the related work in general.

6.2.1 AD Research Models and Prototypes

This section compares AD_{kwik} and its domain model to other research projects in the AD domain, namely the architecture decision description template by Tyree/Akerman [12], an ontology of design decisions by Kruchten et al. [4], Archium by Jansen et al. [54], and the Process-based Architecture Knowledge Management Environment (PAKME) by Babar et al. [55]. All of them provide a domain model for capturing ADs. However, they support different types of tools and decision processes.

Tyree/Akerman [12] propose a template to document ADs, e.g. in Microsoft Word. This template is similar to the domain model used in AD_{kwik}, since it includes attributes like name, problem statement and rationale. In the template, several relationships to artifacts, other ADs, requirements and principles can be captured as well as implications and the status of a decision. However, each template comprises not only the AD description but also the outcome along with the justification. This means, AD and outcome are not distinct elements in the domain model like in the AD_{kwik} domain model. Alternatives are expressed through a list integrated in the template. The template is rich enough to document ADs, e.g. at the end of a project. However, it does not capture history information like author, date of decision and date of rejection. Thus, it does not support the life cycle of an AD which includes discussion about it in team collaboration, permanent processing of new ideas concerning the AD and the decision making process shown in Figure 1.1 on page 6.

Kruchten et al. [4] propose an ontology of ADs and a use case model for an architecture knowledge system. They furthermore evaluate tool support for the use cases. The ontology contains attributes like name, epitome, rationale, state and scope, but also information about the history of an AD. Moreover, an AD can belong to different categories, i.e. can for instance be associated to a particular scope, a state or quality attribute. Like in the architecture decision description template by Tyree/Akerman [12], the proposed ontology does not provide distinct model elements for AD and outcome. Outcome and justification are documented within the AD itself. Alternatives are modeled using an isAlternative dependency between two ADs. This means, each AD represents an alternative. In contrast to the AD_{kwik} domain model, it is not possible to have several outcomes for one AD, e.g. for different design model elements. As mentioned in Section 4.2 and Section 4.4, elements of the ontology are used in the AD_{kwik} domain model. The state management is adopted to the AD_{kwik} domain model. Some of the dependency types, like ‘forbids’ and ‘constrains’, are used in the AD_{kwik} dependency management.
Based on their ontology and use case model, Kruchten et al. evaluated a commercial ontology visualization tool for visualizing ADs and categories. The tool captures ADs and categories in an XML file which represents the proposed ontology. ADs are visualized by bullets which form clusters representing categories. In the tool, users can specify the categories they want to visualize. The tool then displays only those ADs which belong to these categories. For instance, only those ADs which are in state ‘decided’ and belong to the scope ‘organization’ are shown. Based

on the domain model and the database, AD_{kwik} is able to do similar queries. The visualization in the ontology tool does not represent information about ADs themselves. Rather it focuses on representing different views on the whole AD space based on categories specified by the user. The ontology to describe ADs is only partly represented as only specified category names are shown in the tool but no AD description or rationale. However, the representation of AD attributes, like in AD_{kwik}, is important in order to understand the AD space, for rational decision making and asset harvesting. Furthermore, as Kruchten et al. declare themselves, the cluster representation is useful for inspecting the AD space, but not enough to fulfill use cases like impact evaluation and history inspection which are part of the AD_{kwik} concept. Another difference to AD_{kwik} is, that Kruchten et al. do not consider tool support for team collaboration or pre-population. The categorization of ADs is a useful concept for querying different scopes, states or decision drivers. This concept can be adopted in AD_{kwik} in future. Assigning different keywords to ADs can be done in collaboration, which is called *tagging*. QEDWiki already provides the possibility to tag pages, thus ADs with keywords and implements a search mechanism on top of it.

Archium [54] is an Eclipse-based tool with an underlying AD domain model [9]. It directly connects ADs with requirements and implementation components. ADs and components as well as relationships between them are described using an architecture description language. This description is compiled into a graphical representation of components and connectors between components as well as ADs and relationships between ADs and components. One focus of Archium is on modifications of architecture which occur over time. These modifications are represented through an element in the model. Moreover, Archium focuses on the third phase of the decision making process depicted in Figure 1.1 on page 6: enforcement. Not only are ADs directly coupled with components, but also can the description of an architecture partly be generated into Java source code. For few ADs in the decision tree of AD_{kwik}, this could be an interesting additional feature for future implementation. However, it is only possible for few ADs on the asset level which is described in Section 4.2. The usage of Archium is restricted to one project. As ADs are directly captured in source code of the architecture description language, extracting them and providing them to other projects or communities is difficult. Furthermore, the possibility to pre-populate the tool is not described in the Archium publications. Archium does not (yet) support team collaboration. Moreover, decision workflow is not considered.

Process-based Architecture Knowledge Management Environment (PAKME) [55] provides a Web-based knowledge repository similar to AD_{kwik}. However, it has not one domain model for ADs but 25 different templates for different architecture knowledge artifacts including rationale and patterns. Based on these templates, PAKME can collaboratively be populated with explicit and implicit architecture knowledge. Similar to AD_{kwik}, PAKME is based on a Web-based platform which provides different collaboration features. Moreover, PAKME is already populated with a number of patterns and best practices. However, the main difference to AD_{kwik} is, that PAKME does not provide guidance through the decision process. Solely a search mechanism is provided to retrieve content. Dependencies between different assets are not made explicit like in AD_{kwik}. There is no concept of decision workflow.

Table 6.2 summarizes the research work related to AD_{kwik}. It shows, that there are many analogies between AD_{kwik} and related research work. However, AD_{kwik} is the only project, which combines

all features listed in the table. Only one of the approaches, PAKME, supports team collaboration and asset harvesting from projects, which are important concepts in AD_{kwik}. Besides PAKME, AD_{kwik} is the only work, where Web-based tool support is part of the concept. Moreover, none of the approaches support decision workflow and sophisticated dependency management explicitly in a tool. As explained in Sections 4.2.2 and 4.4, the AD_{kwik} concept uses these features to guide the architects through the decision making process. Furthermore, this guidance is also supported by the AD_{kwik} user interface, which is a unique differentiator of AD_{kwik}.

	AD description template (Tyree/ Akerman) [12]	**Kruchten et al. [4]**	**Archium (Jansen et al.) [54]**	**PAKME (Babar et al.) [55]**
Decision workflow	no concept	yes, state management	no concept (all decisions are 'decided' per se)	no concept
Dependency management	only textually noted in template	yes, many dependency types	yes, between ADs and ADs, ADs and components	only textually noted in template
Domain model	yes, but no separation of AD and outcome	rich, but no separation of AD and outcome	yes	25 templates, no separation of AD and outcome
Guidance	no concept	no concept	no concept	no concept (provides search and tagging)
Asset harvesting/pre-population	no concept	no concept	no concept (difficult since ADs very project specific)	yes, pre-population is part of the concept
History of ADs	no concept	yes, author and timestamp	no concept	yes, at least author and date of creation
Collaboration	not supported	not supported	not supported (yet)	yes, underlying collaboration platform
Tool support	no	evaluate tools, make proposals but no own tool (yet)	yes, Eclipse-based (rich client)	yes, Web application
Notes	not tool supported	provides rich domain model; focuses on visualization of whole AD space	focuses on architecture changes and enforcement to implementation level	similar concepts to AD_{kwik}; support of decision process differs

Table 6.2: Summary of AD modeling research approaches.

Besides research work in the AD domain, several tools exist which are in use by architects. The following compares AD_{kwik} to the Architects' Workbench which is such a tool.

6.2.2 Architects' Workbench

The Architects' Workbench (AWB) [13] is an Eclipse based application aiming to support architects in the software development process. It manages knowledge concerning the design of architectures. Unlike other software development and modeling tools, AWB focuses on the work itself, not only on the produced output. AWB possesses a rich user interface providing templates to capture unstructured data and transform it into different models. These templates allow the user

to focus on the content itself instead of the presentation of content. Different viewpoints of the architecture are provided including two viewpoints which cover ADs. ADs are captured using the domain model of the IGSM ARC 100 work product [14], can be interconnected through relationships and organized in a tree. Alternatives can be described and attached to the ADs. The presentation template for the AD domain model is depicted in Figure 6.2.

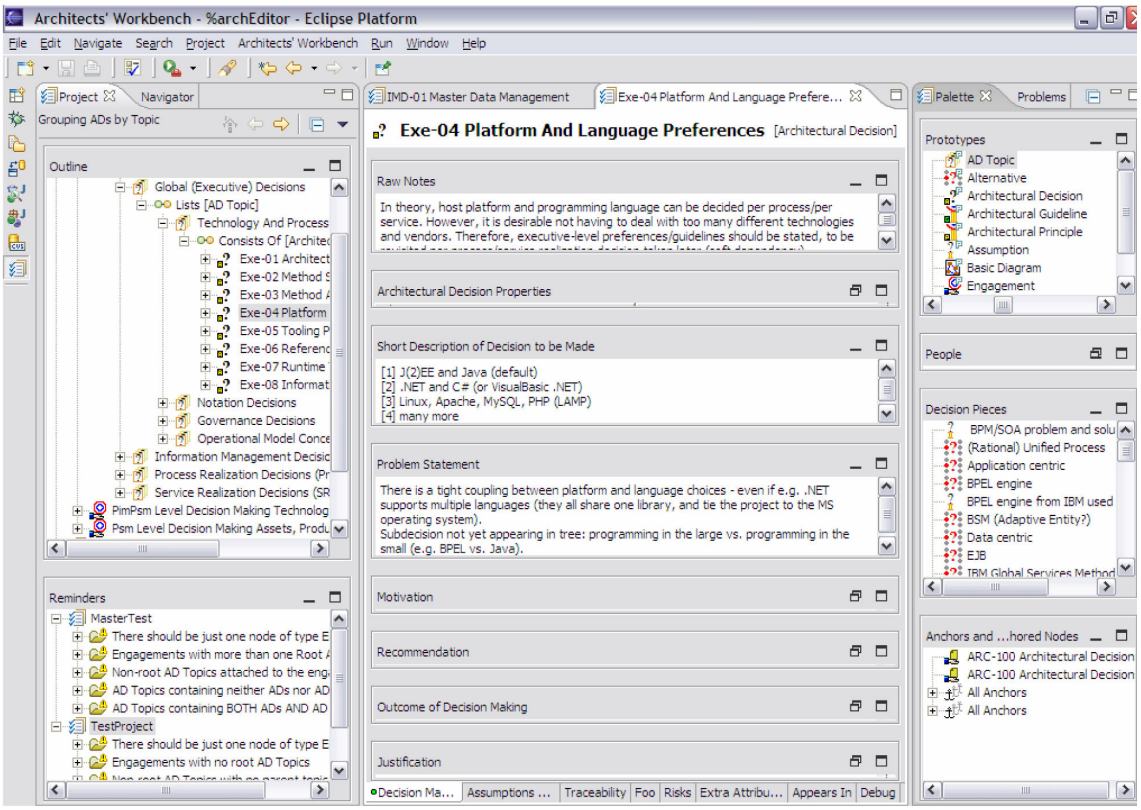


Figure 6.2: User interface of the Architects' Workbench.

The concepts of supporting work instead of results and focus on content instead of presentation resemble the concepts of AD_{kwik}. However, there are several differences between AWB and AD_{kwik}.

As mentioned, AWB is a set of plug-ins built on top of Eclipse using a layered model. Thus, AWB exhibits all shortcomings of a rich client in collaborative content management which are examined in Section 3.2. Unfortunately, content management is implemented through plug-ins which are not usable for collaboration. First, the produced work products are stored in files which contain HTML formatting information. Second, the work products are shared through a simple CVS plug-in. This makes sharing of content difficult. The implementation of the AWB domain layer does not include off-the-shelf support for decision workflow and life cycle. Moreover, collaboration features are not implemented (yet). The concept of alternatives is difficult to use, because the user interface separates alternatives from ADs. This complicates navigation and comparison of alternatives. Moreover, outcome and alternatives are disconnected. Like in most of the research work explained in the previous section, the underlying domain model [14] does not support distinct elements for AD and outcome.

Several advantages of AWB over AD_{kwik} exist as well. AWB provides the possibility to generate architecture documentation from the work products produced during the development process. This documentation can be used to present special views on the architecture to the customer. The

generated documents even can be customized through templates. This feature is part of the AD_{kwik} concept and needs to be designed and implemented in AD_{kwik} in the future.

Moreover AWB allows the architects to capture so called raw notes and transfer them into different types of architecture design models during the process. The note and the model originated from it are linked to provide bidirectional navigation. This feature is not supported by AD_{kwik}, but AD_{kwik} provides a model element for unstructured notes as well, the 'notes' attribute which is introduced in Section 4.2.

Another feature are the *reminders*, which can be seen in the lower left corner of Figure 6.2. These reminders remember the architects of elements in the models which are incomplete or not yet linked models, so called loose ends. This feature is not supported by AD_{kwik} since its domain is restricted to ADs. However, the idea of reminders can be adopted in AD_{kwik} as well. Each architect could have a personalized page which shows the open decisions he is responsible for and reminds him to make them.

To summarize, AWB resembles AD_{kwik} in its basic concepts. However, the realization of the concepts differs fundamentally due to the used framework. Even though the implementation is layered in components, AWB exhibits several shortcomings like missing collaboration features and difficult data sharing.

6.2.3 Pattern Repositories

Pattern repositories are Web sites for collections of design patterns which are most often specific for one domain. Figures 6.3 and 6.4 show the Portland Pattern Repository (PPR) [56], which at the same time is the world's first wiki, and the Yahoo! Design Pattern Library [57].

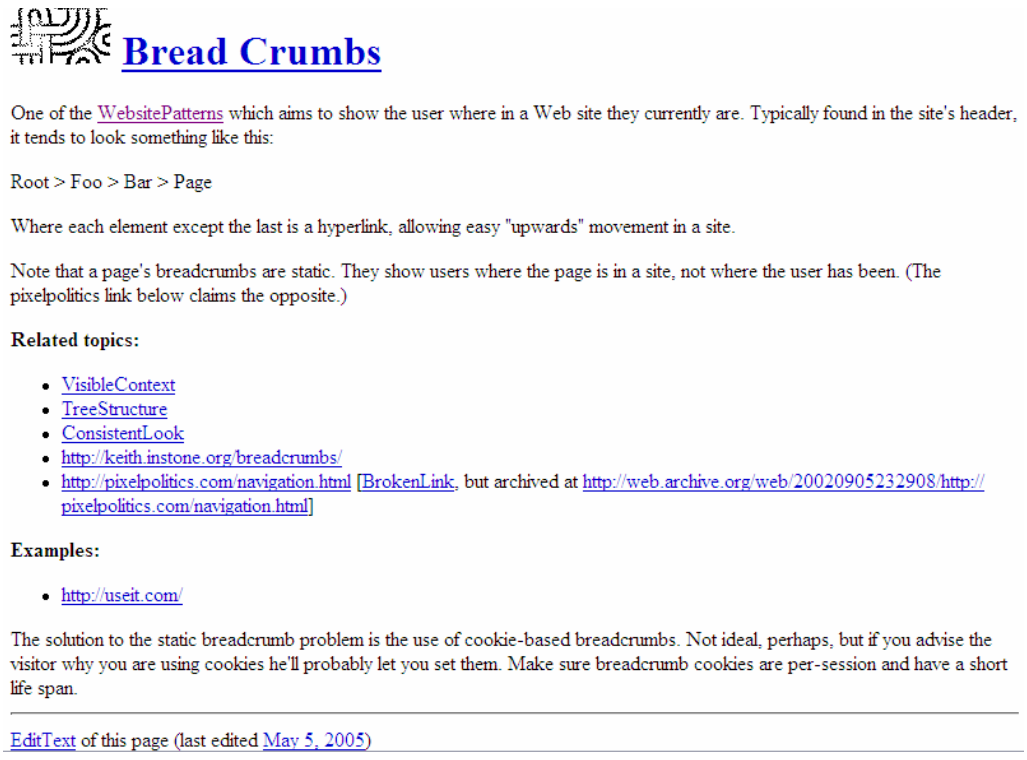


Figure 6.3: User interface of the Portland Pattern Repository [56].

Pattern Repositories provide Web-based descriptions of design patterns which are solutions to common problems in software design. These solutions most often are captured in a structured format, which contains a title of the pattern, a problem description, a solution and an example, e.g. as image, UML diagram, live example or even short video. One important attribute of pattern descriptions are 'forces', which are similar to the decision drivers in the AD domain model. The structuring of pattern descriptions resembles the capturing of ADs in AD_{kwik}. Yahoo! (see Figure 6.4) even organizes the described design pattern in a tree. Furthermore, the linking of similar patterns is done through hyperlinks.

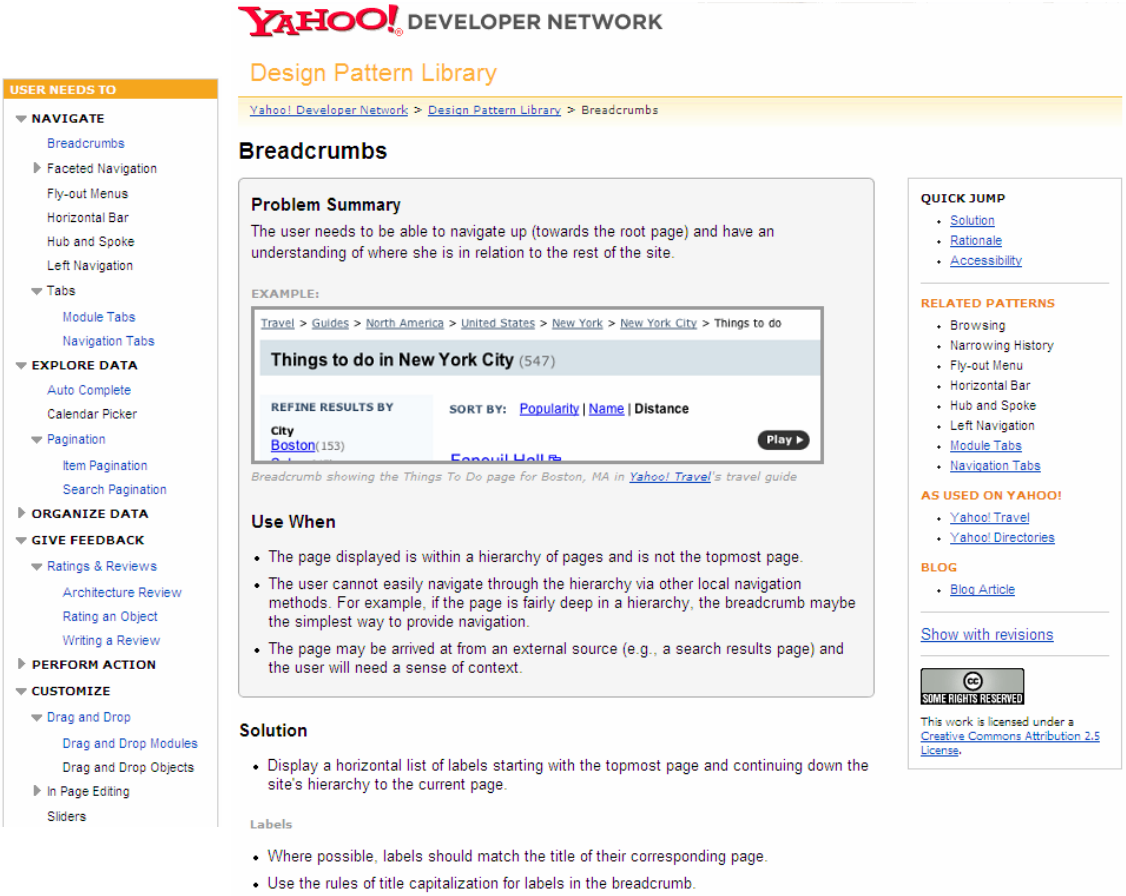


Figure 6.4: Yahoo! Design Pattern Library: Organization of patterns in tree structure (left). Description of a design pattern (right). [57]

However, many pattern repositories do not provide ability to collaborate. An exception is Yahoo! which provides a Weblog with comment features for every pattern. Most pattern repositories describe patterns in their own model. These models usually already contain the solution to the described problem. Through the concept of alternatives, AD_{kwik} provides the choice between several solutions for the problem. Alternatives of one AD therefore could also be seen as patterns for the same problem. Another difference is that AD_{kwik} supports the decision workflow whereas pattern repositories simply present possible solutions. Pattern repositories most often only handle links to related patterns, they do not provide dependency management. Pattern repositories organize the solution space, whereas ADs focus on requirements and orientation in the problem space.

To summarize, pattern repositories resemble AD_{kwik} as they collect possible solutions of a problem in a domain. Differences can be found in the used domain model and especially the implementation of the decision making process which is part of the AD_{kwik} concept.

6.2.4 Enterprise Wikis

As outlined in Section 2.4, enterprise wikis are knowledge management systems operated by communities which typically provide features for better user management, security and content control. Some enterprise wikis offer a mash up mechanism, which turns them into application wikis. Figure 6.5 shows a page of TWiki [29], a typical example for an enterprise wiki. AD_{kwik} indeed is a type of enterprise wiki, as well. Thus, in both AD_{kwik} and enterprise wikis it is simple to create and edit data. Most enterprise wikis provide discussion and collaboration features for each wiki page. Some enterprise wikis, like TWiki and JotSpot [31], can be extended through plug-ins. The administration complexity of wikis is high, since the page hierarchy needs to be maintained to preserve clarity of content. Monitoring and backup need to be done on a regular basis.

Figure 6.5: User interface of the enterprise wiki TWiki [29].

The usability and easy syndication of content as well as the mentioned problems are similar in AD_{kwik}. Differences result from the fact, that conventional enterprise wikis are of general purpose, whereas AD_{kwik} is customized to the AD domain. Therefore, data is displayed in different ways in the wikis. Plain enterprise wikis often use simple HTML templates which can be edited through rich text editors or simple markup languages. The AD content in AD_{kwik} is pre-structured through widgets and the domain model. This eases the design of clearly arranged wiki pages. Enterprise wikis usually do not support any workflow, whereas AD_{kwik} implements the decision workflow.

The integration of user access rights and the possibility of fine granular assignment of user rights is supported by some enterprise wikis. This feature is also needed in AD_{kwik} which, at the time being, only supports two modes: full access to all data or no rights at all. A concept of fine granular rights allows to configure who is allowed to see, edit or decide a particular AD.

In order to support better monitoring of the data, enterprise wikis usually implement a sophisticated history management. AD_{kwik} implements the basics for this management through its versioning mechanism.

To summarize, as AD_{kwik} basically is an enterprise wiki, the shortcomings and advantages of enterprise wikis can also be found in AD_{kwik}. However, AD_{kwik} is customized to the AD domain, which provides better usability through structured content. Furthermore, a unique differentiator of

AD_{kwik} over traditional enterprise wikis is reflected in the pre-population of content of a special domain.

6.2.5 Decision Support Systems

The term Decision Support Systems (DSS) covers a wide range of systems, tools, and technologies. Therefore a comparison to a DSS would go beyond the scope of this thesis. Generally speaking, DSS are systems, which support human beings in decision making. This can happen passively, e.g. through the preprocessing and presentation of data, or actively through suggestions or solutions made by the system. Examples for passive DSS are information systems on base of data warehouses. Clinical DSS (CDSS) are an example for DSS in the health domain. CDSS often include a dynamic knowledge base and inference mechanisms for suggestions based on existing expert knowledge. This makes them similar to AD_{kwik}. At the time being, AD_{kwik} is considered to be a passive DSS. Future work could include a suggestion mechanism, which is based on the dependency management, the decision life cycle and history.

6.2.6 Conclusion

The previous sections analyze related work in research and industry addressing the AD domain as well as technologies which support software development processes and decision making in general.

The comparison showed, that recent research work in the AD domain provides templates for AD modeling, but does not support guidance through the decision making process. Underlying dependency management and decision workflow is not supported in an adequate way to cover the decision process. AWB addresses the AD domain, but has a different implementation approach which makes it difficult to support team collaboration. Pattern repositories use a similar implementation approach in that they are Web-based. However, they only address one part of the AD domain: They capture knowledge about software patterns, which are treated as alternatives in AD_{kwik}. Decision processes are not supported explicitly. Enterprise wikis follow the same implementation but are not customized to any domain. Decision support systems and software modeling tools usually use the same implementation approach as AWB, but have the same shortcomings as enterprise wikis and typical decision support systems: They are not customized to the AD domain and most often do not support pre-population with content. All of the described applications have underlying content repositories.

The following list summarizes the evaluated related work shortcomings.

- Approaches addressing the AD domain do not support collaboration features (yet).
- Approaches supporting collaboration are not customized to the AD domain and thus do not support a suitable domain model or pre-positioned content. However, some systems which support knowledge management could be customized to support the domain (e.g. special enterprise wikis).

- Approaches supporting knowledge management in general are not customized to the AD domain and thus do not support a suitable domain model and most often do not feature collaboration.
- AD domain models and ontologies are not rich enough to support guidance through the decision making process.
- Related software development tooling does not yet support architectural decision making processes sufficiently.

Especially AD_{kwik}'s support of decision making processes and the guidance of architects through these processes as well as collaborative AD modeling and knowledge management are a unique combination.

6.3 First User Feedback

Since December 2006, the AD_{kwik} concept and prototype were presented in workshops to several persons of the target audience as well as technical deciders in and outside IBM. AD_{kwik} already is in use within one industry project. Interviewing the architects gave feedback concerning concept and prototype as well as wishes for new features. Moreover, one architect conducted a usability and function test of AD_{kwik} in comparison with Microsoft Word ARC 100 tables and AWB. Since prototype and pre-provisioned content intertwine, the feedback described in the following concerns both the tool and the content.

Initial user feedback regarding value and usability of AD_{kwik} shows, that users appreciate that all knowledge required during architectural decision making can be conveniently located in a single place. Furthermore, they found useful that the system comes with a set of initial AD content. For instance, one user reported, that the effort for the creation of a SOA principles deliverable decreased from eight to five person days because 13 out of 15 required decisions were present in AD_{kwik} and could be reused. Despite the large size of the decision space, early users reported to be productive without major training efforts. They appreciate the depth of detail of the domain model and the usage of a wiki as concept. In comparison with AWB and ARC 100 AD_{kwik} showed benefits in collaboration and team support.

AD_{kwik} leverages Web 2.0 application wiki technology whose user experience perceived to be compelling. Users looked favorably upon search and browsing through the decision tree in the content explorer and the overall navigation. Especially in comparison with AWB and ARC 100 usage, AD_{kwik} scored well with regard to navigation. The incorporation of alternatives into the description of an AD like shown in Section 5.3.3 and the displaying of the most important content on one screen without need to scroll was appreciated. Subsecond response times in first load tests with a single user were reported.

The first adopters gave constructive criticism regarding information architecture, usability and functionality. They suggested several improvements and new features:

- Refactoring is not supported well in AD_{kwik}, especially in comparison with AWB. Moving ADs in the decision tree can only be done through typing the path to the new location. A concept like dragging and dropping of ADs in the decision tree is needed for refactoring.
- The AWB import is rather slow and should be improved.
- For first adopters, the menu structure is not easy to understand, especially the difference between AD_{kwik} and QEDWiki menu. Restructuring of the menu and further explanation in the user's guide are required.
- The edit button for the AD should be placed on top of the page to be easier to find.
- Export should be implemented, as well as decision tree copying within the wiki.
- The user interface should display the outcome(s) separately from the AD description, because of the 1:n relationship between AD and outcome.
- To improve the asset harvesting, project teams should have the possibility to comment their decisions in the end of a project. They could explain whether they would decide again for particular ADs and indicate the reasons if not.
- Decision tree pruning, i.e. the automatic deletion of obsolete ADs, should be implemented on top of dependency management and decision workflow to further improve guidance through the decision making process.

Users criticism also regarded challenges of modeling a large and complex decision space facing a high degree of change: ADs and alternatives become available almost on a daily basis. When aiming for completeness, this could lead to several thousand ADs with numerous alternatives and dependencies. This complexity can degrade the usability. However, experienced practitioners report, that they prefer to be made aware of the complexity and to have a system that manages it, rather than to let the knowledge remain tacit and unmanaged.

Based on the feedback of the initial evaluation, AD_{kwik} will be further improved. Usability studies on a larger scale are planned, in which several architects from the first feedback round will volunteer as testers.

6.4 Summary of Evaluation

The previous sections evaluate AD_{kwik} against requirements, compare them to related work and listed first user feedback. The analysis of AD_{kwik} against the requirements shows, that all requirements are met by the conceptual design of AD_{kwik}. The key requirements are already implemented in the prototype.

The comparison to related work reveals, that AD_{kwik} provides a unique combination of Web 2.0-based collaboration systems and the AD domain. As outlined in Section 6.2.6, key differentiators from related work are the guidance through the decision making process of architects through dependency management and decision workflow as well as collaborative AD knowledge management.

First tests with the user community are encouraging: The interviewed architects appreciate the usability of the system as well as the concepts realized. Constructive criticism and wishes for new features and change cases show, that AD_{kwik} is well on the way to being accepted and used by the architects community.

The next chapter reflects on the methodology used and lessons learned during the course of the thesis. Furthermore, it outlines future work which results from the evaluation in this chapter. It then concludes the thesis with a summary.

7 Reflection, Outlook and Summary

This chapter completes the thesis with a short reflection regarding the methodology used, the challenges faced and the lessons learned during the development. Then some suggestions for future work are made. Finally, a summary concludes this work.

7.1 Methodology and Lessons Learned

The work for this thesis was conducted in three phases, each taking two months:

1. Evaluation, Analysis and Design
2. Implementation
3. Documentation and User Tests

The phases were loosely planned beforehand, to keep track of the project status.

Evaluation, Analysis and Design

The first two months of the thesis were used to understand and collect information about the domain, gather requirements, analyze candidate technologies and existing tools, implement simple prototypes, and finally design the solution. In this first phase the different steps, especially capturing of requirements through discussions were conducted iteratively in an agile approach. Several work products of the IBM Global Services Method, like the component and the use case model, were produced and UML diagrams created. The decisions made in this phase had to be well considered since they influenced the following phases. The decisions were captured and documented in AD_{kwik} once the first prototype was available to demonstrate the usability of the concept and improve it by own experience.

The first prototype of the user interface was designed using Microsoft Power Point, as this turned out to be faster and more flexible than implementing it in HTML. A first user test with an IT architect on the base of a scenario provided valuable feedback concerning the design and usability of the user interface.

The most tedious activity during this phase was learning and testing of QEDWiki to be able to decide whether to use it. As most of the documentation of QEDWiki is addressed to the user of situational applications and not the developers, getting familiar with the framework internals was difficult. As the first version of QEDWiki finally was released end of October 2006, the decision

was made in favor of it. Compared to its main competitors, Eclipse RCP and implementing the application from scratch, the implementation of a usable prototype during two months were assessed to be feasible. During the evaluation phase, several other wiki frameworks were reviewed. This includes TWiki [29], an enterprise wiki which can be extended through plug-ins. TWiki is implemented in Perl, the architecture is not open, solely an API is provided. Thus, layering and object-oriented programming is difficult to accomplish with TWiki. This was the reason why not choosing it. Another evaluated wiki was Makna [30], a semantic wiki. Semantic annotation of pages is useful for managing relationship between pages and search functionalities. However, this wiki does not provide an extension mechanism, which was needed to implement domain logic, for instance decision workflow.

During the QEDWiki evaluation, the mash up mechanism of QEDWiki was analyzed which is the main focus of the QEDWiki developers. QEDWiki widgets need to be parametrized in the user interface before usage. This parametrization is difficult to use, and too many commands exist, which are not structured in an intuitive way. The analysis resulted in the conclusion, that the usability of the mash up mechanism to build situational applications is not yet mature enough in order to apply it in productive environments. The idea, to use dragging and dropping of ADs from the QEDWiki palette into the wiki pages therefore was rejected and the wiki was totally customized to ensure the usability of the application. Nevertheless, mashing up of AD_{kwik} pages with ADs and AD topics is possible, as they are implemented as widgets. However it is not recommended, because it is easier to add them via the AD_{kwik} navigation menu. Users of AD_{kwik} therefore do not have to understand the complex QEDWiki mash up mechanism in addition to AD_{kwik}.

Another challenge was to find the right granularity for the design of the domain model, the dependency management and the decision workflow. This challenge was faced by interviewing architects of the target audience and examining different domain models from research and industry like Kruchten et al. [4], Tyree/Akerman [12] or the ARC 100 work product [14].

Implementation

The next two months were used to implement the proposed solution. As QEDWiki still was a technical prototype, there was no community and not much documentation like tutorials on how to extend it. This raised difficulties in learning the extension mechanism and the provided API.

Especially for the development and evaluation of the user interface, the captured use cases turned out to be very valuable. Capturing of the technical requirements separately from the use cases resulted in an easier implementation of the domain and data source layer, since the requirements could be used directly to build function blocks of the solution.

The most significant challenge in this phase was the mapping of the conceptual design to the technical layer and QEDWiki since this task requires experience. To assist this task, the IGSM Component Model work product [58] was adopted. Furthermore, layering the QEDWiki architecture into presentation, domain and data source layer helped in understanding how to extend QEDWiki with own logic.

Documentation and User Tests

The third phase comprised creation of a user's and developer's guide, writing of the thesis and first user tests. Getting started with LaTex is not difficult, however, several details like including pictures or tables took their time. In this phase, the capturing of ideas and discoveries during earlier phases turned out to be useful in order to remember and document the analyzed and rejected technologies.

Furthermore, as the application was tested through users, bugs, implementation shortcomings and change cases were discovered and many ideas for new features were developed. These tests showed that large-scale user tests are essential for the usability of concept and tool. The next section covers these change cases and ideas.

7.2 Future Work

The evaluation of AD_{kwik} in Chapter 6 revealed several fields of future work shortly summarized in the following:

- Generation of templated documentation from AD content, e.g. for customers
- History and rollback functionality based on the already provided versioning mechanism
- Suggestion/recommendation algorithms
- Provide constraint management, e.g. for incomplete AD descriptions or not yet decided ADs, to support reminders for architects like in the AWB
- Fine granular user rights system and security features
- Mapping of ADs to the implementation level (enforcement)
- Implementation of advanced search functionality
- Implementation of use cases anonymization and export
- Tagging of ADs
- Performance and scalability improvements
- Possibility to comment on decisions at the end of a project
- Larger-scale usability tests

Furthermore several other fields of future work are evaluated and explained in the following.

The user interface of the prototype has several shortcomings. First of all, it is not possible to define several outcome instances for one AD. If for instance several Web services should be implemented, the AD 'Web services style' has to be decided for each Web service. At the time of writing, for each outcome a new page with the same AD must be created. This leads to an unnecessary degradation of the usability. Therefore, a solution is needed to capture several outcomes on the same page. This would also show, that the outcome is part of the decision making phase and not the decision identification.

Another feature concerning the user interface is more elaborated guidance, e.g. through roadmaps on the start page, which show the user next steps, or simplified trees. This feature could address both the novice and the experienced architects according to their needs. Furthermore the user interface needs some further effort. This could include implementing a drag and drop mechanism in the decision tree to move decision entries.

QEDWiki's collaboration features can not replace real communication and decision making. For this reason, other collaboration tools can be integrated, like an Internet forum, which allow further structuring of the discussion. The integrated collaboration tools have to be carefully evaluated and wisely chosen, because not all tools support structuring of content.

On the domain layer, there are several ideas for the future as well. The decision workflow needs to be extended to support a justification when rejecting a decision. This results in a more detailed and traceable decision log.

Based on the data structure of the dependencies, several features can be implemented for an improved dependency management. One idea is to use the semantics behind a dependency to find not relevant ADs and hide them from the user. This mechanism is also referred to as *decision tree pruning*. It makes the repository more dynamic. To give an example, if AD 'Language and Platform Preferences' is decided in favor of '.Net and C#', all Java-relevant decisions like 'Java JDK' are hidden. Another scenario could be to simulate what-if situations. This could help architects to understand the impacts of a special decision. To switch between simulation and real decision making, the user could configure his user account. This configuration possibility could also be used to specify if the user wants to hide or show irrelevant ADs. This idea is shown in Figure 7.1.

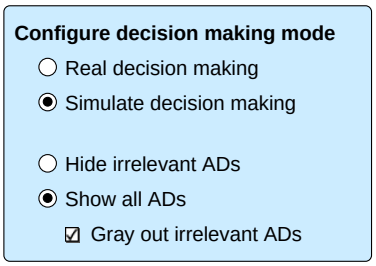


Figure 7.1: Possible user settings configuration parameters.

The integration of AD_{kwik} into the enterprise environment needs to be addressed, as well. This could happen through providing a Web service interface. Further planned features address integration of modeling software like IBM WebSphere Business Modeler [59] which capture requirements in form of business processes and services. These processes and services are related to the decision tree: For each process and service a set of ADs need to be decided. An integration includes a concept of mapping required processes and services to ADs and a generation of the customized decision tree for the requirements.

7.3 Summary

This thesis introduced AD_{kwik}, a tool-supported concept for collaborative architectural decision modeling, making and knowledge management and presented the AD_{kwik} prototype implementation. The first chapter gave an introduction into the AD domain. It explained that ADs are important in the software development process as they hold architectural knowledge that would remain tacit if not explicitly captured. Thus, ADs need a first-class representation unlike in state of current practices. Moreover, the introduction in Chapter 1 presented the decision making process which consists of decision identification, decision making and decision enforcement. It furthermore outlined why tool support on top of AD modeling is needed: Decision making is done in collaboration, AD models can be reused through pre-population and practitioner communities steadily improve and add knowledge.

Chapter 2 introduced several technologies and paradigms used in this thesis. First of all it explained the technique of architecture layering and the main differences between rich and thin clients. The next section introduced several collaboration applications like Weblog, instant messaging and message boards. A short introduction into Web 2.0 paradigms, wikis and situational applications clarified that there is also another client paradigm in addition to rich and thin client: the Rich Internet Application (RIA). RIAs can be realized with Ajax, which was explained in last section of the chapter.

The requirements analysis concerning the approach, summarized in Chapter 3, pointed out that a content repository is needed, which supports a usable user interface, collaboration features, the decision process, as well as AD modeling. Chapter 3 furthermore presented two different types of client frameworks: Eclipse RCP as a typical rich client framework and QEDWiki as a RIA framework. The evaluation of these frameworks showed that the framework component and layer structure is quite similar. Thus, the customization of the frameworks can be implemented in similar ways. In addition, the evaluation clarified that user interfaces of RIAs can be as sophisticated as rich client user interfaces.

Based on the requirements and the evaluation, Chapter 4 developed a conceptual design. The analysis of frameworks in Chapter 3 led to the decision to base the concept on QEDWiki, a Web 2.0 situational application wiki. The reasons for this choice were the sophisticated user interface of RIAs, the advantages of thin clients and the layered architecture with extension mechanism which is comparable to rich client frameworks. On this base, the chapter explained the different elements of the concept: The domain model underlies the application and includes a dependency management which handles topic, time and outcome dependencies between ADs. Data is persisted in a database, using the Active Record pattern. Moreover, versioning with the Edition pattern is part of the concept. The decision workflow to support the decision making process is based on state management of outcomes. QEDWiki features are used to support collaboration. Finally, the user interface concept was explained which comprises information architecture, site layout and the representation of the domain model.

Chapter 5 explained how the concept was implemented in a prototype. It showed that the domain model was mapped to the database as data model with foreign key relationships. Moreover, it

explained how the Active Record pattern was used to map objects to the database. The chapter then outlined how versioning and AWB import was realized, how the user interface was implemented and looked like. Finally, it showed statistical numbers of the implementation and the tools used throughout the AD_{kwik} development.

Chapter 6 evaluated the concept and the implementation. The evaluation against the requirements of Chapter 3 showed, that all requirements are met by the concept and the key requirements are implemented in the prototype. The comparison to related work showed that ongoing projects exist which focus on the AD domain. However, most of the related work focuses either on the AD domain or on collaboration tool-support. None of the related work sufficiently supports the decision making process. First user feedback was positive but also raised discussions for future work. It showed that the prototype is usable already, however, due to time restrictions, the implementation is still in an early state and needs improvements.

Chapter 7 outlined the methodology used and lessons learned during AD_{kwik} development. It furthermore suggested fields for future work, like improvements of the user interface, decision tree pruning and large-scale user tests.

If AD_{kwik} is accepted by the users as complementary tool to standard software architecture development tooling e.g. for UML modeling – feedback from the target audience indicates this – it is a promising concept to make architectural decisions first class entities in the architecture design process. As illustrated in Chapter 1, it then promises to provide benefits like time and budget savings through experience and knowledge reuse, acceleration of architecture design project steps, improvement of decision making and finally producing architectures of higher quality.

Bibliography

[1] P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, pp. 42–55, 1995.

[2] P. Kruchten, *The Rational Unified Process: An Introduction*. Addison-Wesley, 2003.

[3] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[4] P. Kruchten, P. Lago, and H. van Vliet, "Building up and Reasoning about Architectural Knowledge," *2nd International Conference on the Quality of Software Architectures (QoSA)*, vol. LNCS 4214, pp. 43–58, 2006.

[5] J. Lee and K.-Y. Lai, "What's in Design Rationale?" *Human Computer Interaction*, vol. 6, pp. 251–280, 1991.

[6] A. MacLean, R. Young, V. Bellotti, and T. Moran, "Questions, Options, and Criteria: Elements of Design Space Analysis," *Human-Computer Interaction*, vol. 6, pp. 201–250, 1991.

[7] J. Bosch, "Software Architecture: The Next Step," *Software Architecture, First European Workshop (EWSA)*, vol. 3047 of LNCS, pp. 194–199, Springer, May 2004.

[8] O. Zimmermann, J. Koehler, and F. Leymann, "The Role of Architectural Decisions in Model-Driven SOA Construction," 2006, OOPSLA 2006 (Portland, Oregon, USA, October 21 - 26, 2006).

[9] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," pp. 109–118, 2005.

[10] O. Zimmermann, T. Gschwind, J. Küster, and N. Schuster, "Reusable Architectural Decision Models for Enterprise Application Development," *Submitted to QoSA*, 2007.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, 1995, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[12] J. Tyree and A. Akerman, "Architecture Decisions: Demystifying Architecture," *IEEE Software*, vol. 22, pp. 19–27, 2005.

[13] S. Abrams, B. Bloom, P. Keyser, D. Kimelman, E. Nelson, W. Neuberger, T. Roth, I. Simmonds, S. Tang, and J. Vlissides, "Architectural thinking and modeling with the Architects' Workbench," in *IBM Systems Journal*, vol. 45, no. 3, 2006. [Online]. Available: http://www.research.ibm.com/journal/sj/453/abrams.html

[14] IBM Corporation, "Architectural Decisions," Aug. 2002, Work Product Description, Unique ID: ARC 100. Version: 4.1.1.

[15] Carnegie Mellon University, "CMMI for Development, Version 1.2," Tech. Rep., 2006, CMU/SEI-2006-TR-008. [Online]. Available: http://www.sei.cmu.edu/cmmi/cmmi.html

[16] R. Haas, "Usability Engineering in der E-Collaboration. Ein managementorientierter Ansatz für virtuelle Teams," Ph.D. dissertation, 2004, Deutscher Universitäts-Verlag, ISBN 3-8244-2175-5.

[17] P. Gongla and C. Rizzuto, "Evolving communities of practice: IBM Global Services experience," *IBM Systems Journal*, vol. 40, 2001.

[18] A. Tang, M. A. Babar, I. Gorton, and J. Han, "A Survey on Architecture Design Rationale," Swinburne University of Technology and NICTA, Tech. Rep., 2005. [Online]. Available: http://www.swin.edu.au/ict/research/technicalreports/2005/SUTICT-TR2005.02.pdf

[19] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[20] D. Trowbridge, D. Mancini, D. Quick, G. Hohpe, J. Newkirk, and D. Lavigne, *Enterprise Solution Patterns Using Microsoft .NET*. Microsoft Corporation, 2003. [Online]. Available: http://msdn2.microsoft.com/en-us/library/ms998469.aspx

[21] P. Wilson, "Computer Supported Cooperative Work (CSCW): Origins, concepts and research initiatives." *Comp. Networks ISDN Syst.*, vol. 23, pp. 91–95.

[22] T. Davenport and L. Prusak, *Working Knowledge: How Organizations Manage What They Know*. Harvard Business School Press, Boston, MA, 1998.

[23] T. O'Reilly, "What is Web 2.0: Design Patterns and Business Models for the Next Generation of Software," O'Reilly Media, Inc., Sep. 2005.

[24] J. J. Garrett, "Ajax: A New Approach to Web Applications," 2005. [Online]. Available: https://adaptivepath.com/publications/essays/archives/000385.php

[25] "Adobe Flash." [Online]. Available: http://www.adobe.com/products/flash/flashpro/

[26] A. Ebersbach, M. Glaser, and R. Heigl, *Wiki – Web Collaboration*. Springer Berlin Heidelberg, 2006.

[27] "WikiWikiWeb." [Online]. Available: http://c2.com/cgi/wiki

[28] "Wikipedia." [Online]. Available: http://en.wikipedia.org/wiki/Wikipedia:About

[29] "TWiki." [Online]. Available: http://twiki.org/

[30] "Makna." [Online]. Available: http://www.apps.ag-nbi.de/makna/wiki/About

[31] "JotSpot." [Online]. Available: http://www.jotspot.com/

[32] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol HTTP/1.1," *RFC2616*, 1998. [Online]. Available: http://www.w3.org/Protocols/rfc2616/rfc2616.html

[33] J. Nielsen, *Usability Engineering*. Morgan Kaufmann, San Francisco, 1994.

[34] D. Bredemeyer and R. Malan, "The Role of the Architect," 2006, Architecture Resources. For Enterprise Advantage. [Online]. Available: http://www.bredemeyer.com

[35] M. Galic, J. Adams, J. A. Bell, R. Disney, V.-M. Kanerva, S. Matulevich, K. Rebman, and P. Spaas, *Patterns: Applying Pattern Approaches Patterns for e-business Series*. IBM Redbooks, 2003.

[36] "Eclipse Rich Client Platform (RCP)." [Online]. Available: http://www.eclipse.org

[37] "QEDWiki." [Online]. Available: http://services.alphaworks.ibm.com/qedwiki/

[38] "OSGi (Open Services Gateway initiative) Alliance." [Online]. Available: http://www.osgi.org

[39] "Zend Framework." [Online]. Available: http://framework.zend.com/

[40] "JavaScript Object Notation (JSON)." [Online]. Available: http://json.org/

[41] G. Hohpe and B. Woolf, *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[42] "Concurrent Versions System (CVS) Eclipse Plug-In." [Online]. Available: http://www.eclipse.org/eclipse/platform-cvs/

[43] "Apache Derby." [Online]. Available: http://db.apache.org/derby/

[44] L. Cheng, S. Hupfer, S. Ross, and J. Patterson, "Jazzing Up Eclipse with Collaborative Tools," 2003.

[45] B. Daum, *Das Eclipse-Codebuch*. dpunkt.verlag, Jan 2006.

[46] J. Miller and J. Mukerji, "MDA Guide Version 1.0.1," Object Management Group doc.omg/2003-06-01, June 2003. [Online]. Available: http://www.omg.org/cgi-bin/doc?omg/03-06-01

[47] F. Anderson, "A Collection of History Patterns," Aug. 1998.

[48] L. Rosenfeld and P. Morville, *Information architecture for the World Wide Web*. O'Reilly & Associates, Inc., 1998.

[49] B. Lida and B. Chapparo, "Breadcrumb navigation: Further investigation of usage." Department of Psychology, Wichita State University, Tech. Rep., 2003.

[50] "Dojo Toolkit." [Online]. Available: http://dojotoolkit.org/

[51] "Apachefriends XAMPP." [Online]. Available: http://www.apachefriends.org/en/xampp.html

[52] "PHPeclipse plug-in." [Online]. Available: http://www.phpeclipse.de

[53] "Rational Software Architect." [Online]. Available: http://www-304.ibm.com/jct03002c/software/awdtools/architect/swarchitect/

[54] A. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer, "Tool support for Architectural Decisions," *Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA)*, Jan. 2007.

[55] M. Babar, I. Gorton, and R. Jeffery, "Capturing and using software architecture knowledge for architecture-based software development," *Fifth International Conference on Quality Software QSIC*, pp. 169–176, 2005.

[56] "Portland Pattern Repository." [Online]. Available: http://c2.com/ppr/

[57] "Yahoo! Design Pattern Library." [Online]. Available: http://developer.yahoo.com/ypatterns/

[58] IBM Corporation, "Component Model," July 2004, Work Product Description, Unique ID: ARC 108. Version: 4.1.2.

[59] "WebSphere Business Modeler." [Online]. Available: http://www-306.ibm.com/software/integration/wbimodeler/

Index

Acknowledgments

I would like to thank all people who helped during the course of this thesis. Special thanks go to my supervisors, Olaf Zimmermann and Prof. Walter Kriha, for their support and many fruitful discussions. Furthermore, I would like to thank my manager Dr. Jana Koehler for her support and Felix Feger, Christian Gerth and Michel Zedler for proofreading this thesis. I very much appreciate the opportunity to write my thesis on a research oriented topic at the IBM Zurich Research Laboratory. I am grateful to all members of the Business Integration Technologies group as well as Verena Schlegel, Bernard Clark, and Daniel Frank for constructive comments.

Appendix

Appendix A: List of Abbreviations

Appendix B: Use Cases

- This appendix includes an overview of all AD_{kwik} use cases captured during the solution outline phase. Furthermore, it includes selected use cases captured in the use case template of the IGSM Use Case Model work product. Not depicted use cases were described analogously.

Appendix C: User Interface

- This appendix includes a selection of user interface prototypes created in Microsoft PowerPoint during the solution outline and design phase of AD_{kwik}.

Appendix D: AD_{kwik} Reference

- This appendix includes a quick-reference for the AD_{kwik} user interface.

Appendix E: AD_{kwik} Components

- This appendix includes selected descriptions of the AD_{kwik} architecture components captured in the component template of the IGSM Component Model during the solution outline phase of AD_{kwik}.

Appendix F: AD_{kwik} Decisions

- This appendix includes screenshots of selected ADs of AD_{kwik} captured in AD_{kwik}.

Appendix G: First User Tests

- This appendix includes the first tests: a user interface test from December 2006 and a usability and function test from March 2007.

Appendix A: List of Abbreviations

AD	Architectural Decision
AD_{kwik}	Architectural Decision knowledge Web interchange kit
Ajax	Asynchronous JavaScript and XML
API	Application Programming Interface
AWB	Architects' Workbench
CDSS	Clinical Decision Support System
CMMI	Capability Maturity Model Integration
CoP	Community of Practice
CRUD	Create, Read, Update, Delete
CSCW	Computer Supported Cooperative Work
CVS	Concurrent Versioning System
CSS	Cascading Style Sheets
DSS	Decision Support System
FAQ	Frequently Asked Questions
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IBM	International Business Machines
IGSM	IBM Global Services Method
JSON	JavaScript Object Notation
NFR	Non-Functional Requirements
OSGi	Open Services Gateway initiative
PHP	Hypertext Preprocessor
QEDWiki	Quick and Easily Done Wiki
RCP	Rich Client Platform
RIA	Rich Internet Application
RUP	Rational Unified Process
SOA	Service Oriented Architecture
SWT	Standard Widget Toolkit
TR	Technical Requirement
UC	Use Case
UML	Unified Modeling Language
XAMPP	Windows/Linux/Mac OS X/Solaris, Apache, MySQL, PHP, Perl
XML	eXtensible Markup Language

Appendix B: Use Cases

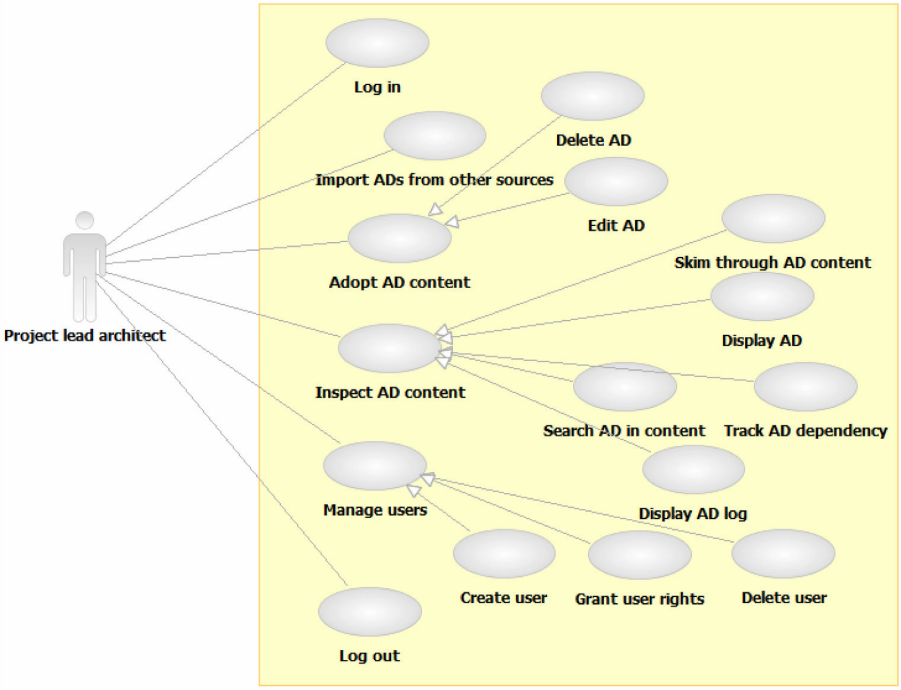


Figure 7.2: Use cases in the project initiation phase.

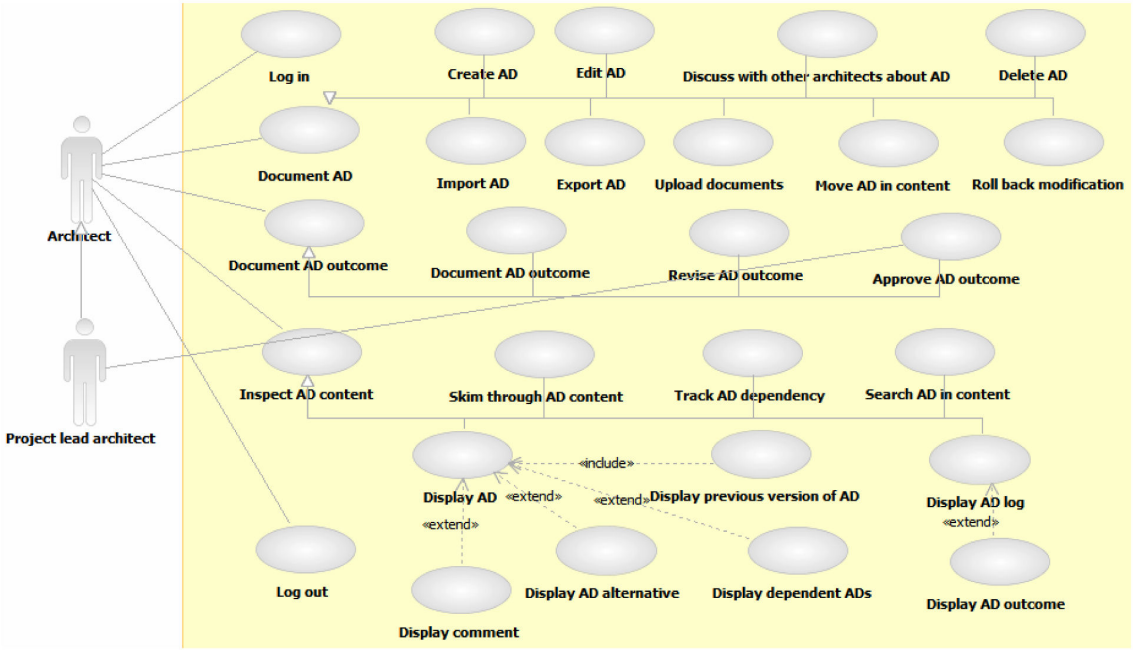


Figure 7.3: Use cases in the solution outline and design phase.

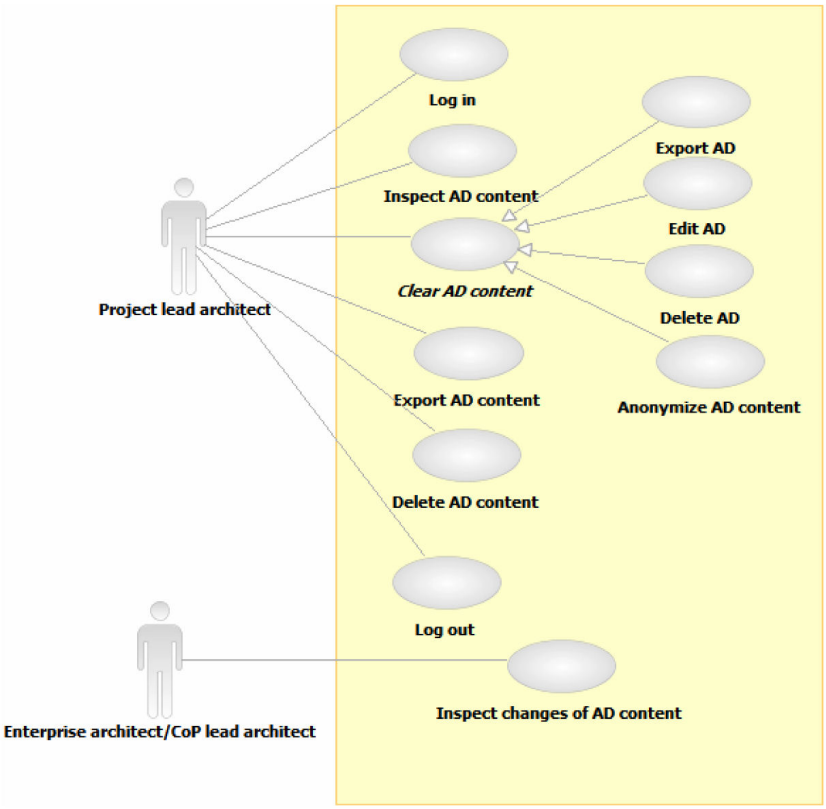


Figure 7.4: Use cases in the project closure phase.

Use Case no. 1	**Import ADs from other sources**
Subject Area	Phase: Project Initiation Phase, Identification
Business Event	Enterprise Architect/CoP Lead Architect releases new version of AD content to work with, Project Lead Architect has to import it.
Actor(s)	Project Lead Architect
Use Case Overview	The architect imports new decision content version from import file.
Preconditions	Authentication and authorization System in "clean" state (no decision content) Start view Technical requirements: input field (for file name) submit button
Termination Outcome	**Condition Affection Termination Outcome**
1. AD content is loaded into system	
2. error, AD content partly loaded	File is corrupt Storage is full
3. error, nothing loaded	No file at given location No rights for reading the file File is corrupt, not a valid import file Database not running No database connection
Use Case Description	+++ The Project Lead Architect tells the system to open the AD Import user interface. The system answers with the requested view. The Project Lead Architect then inserts the path to the new version of the AD content (e.g. .awb file) into the input field and sends it to the system by pressing the submit button. The system traverses the file and stores all ADs, AD topics, dependencies and alternatives. The system then creates the AD content structure (needed for the view). The Project Lead Architect gets an “ok”-message. After that the architects can work with the AD content.

	- - - The Project Lead Architect tells the system to open the AD Import user interface. The system answers with the requested view. The Project Lead Architect then inserts the path to the new version of the decision content (e.g. .awb file) and sends it to the system by pressing the submit button. The system fails to traverse the file and gives back an error message to the Project Lead Architect.
Inputs Summary	Path to import file
Outputs Summary	ok or error message

Table 7.1: Use case Import ADs from other sources

Use Case no. 2	**Create AD**
Subject Area	Phase: Solution Outline and Design Phase, Identification
Business Event	A new AD was identified which is not in the AD content yet
Actor(s)	Architect, Project Lead Architect
Use Case Overview	The architect creates a new record of AD in the content.
Preconditions	AD content Entry view Authentication and authorization Existing AD content (UC Import AD content executed) Entry View Technically: Input form (meta model conform) with submit button
Termination Outcome	**Condition Affection Termination Outcome**
1. The AD is stored in system (new record in database)	Unique name of AD provided
2. Failure, AD cannot be stored	database not running no database connection
Use Case Description	+++ The Architect tells the system to open the AD Creation user interface. The system answers with the requested view. The Architect fills in the required parameters (AD name) and sends this as a message to the system by pressing the submit button. After ensuring that no AD with the same name does exist in the system, the system adds some more parameters (user name, timestamp, version no, default values) and stores an AD record in the database. It then creates an AD object and opens the dialog for editing the AD. - - - The Architect tells the system to open the AD Creation user interface. The system answers with the requested view. The Architect fills in the required parameters (AD name) and sends this as a message to the system by pressing the submit button. The system adds some more parameters (user name, timestamp, version no, default values) and fails to store the record in the database. The system answers the Architect with an error message which includes the reason of the failure (if known).
Inputs Summary	AD name, description elements (cp. domain model), user ID, date and time of input
Outputs Summary	Ok or error message
Use Case Notes	Input could be in rich text or HTML.

Table 7.2: Use case Create AD.

Appendix C: User Interface

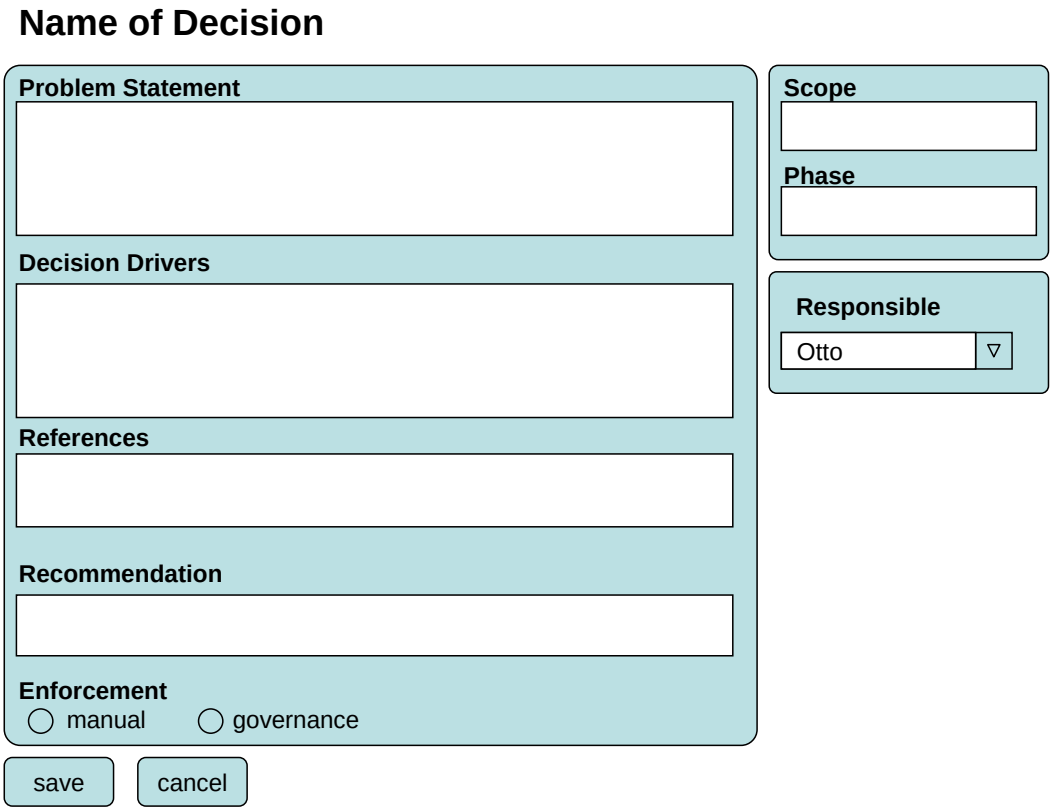


Figure 7.5: Edit AD user interface.

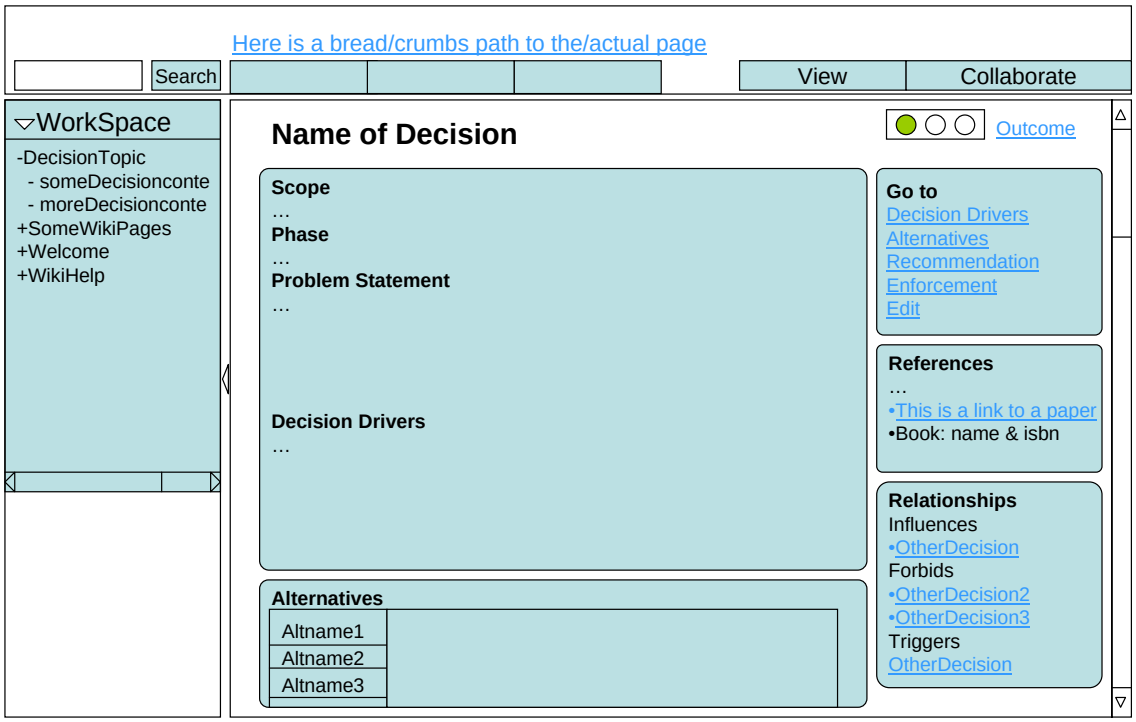


Figure 7.6: Navigation concept.

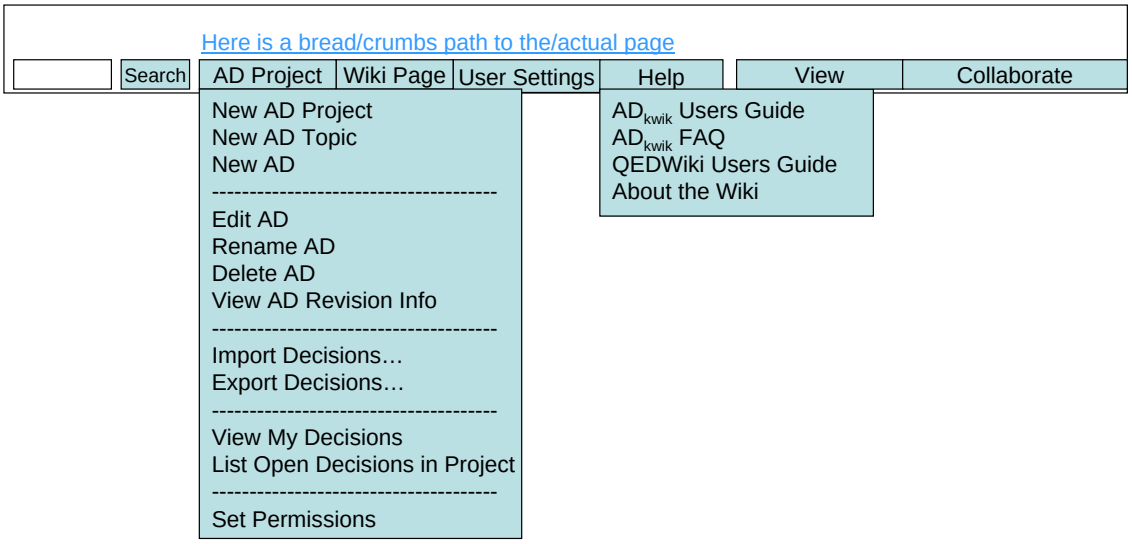


Figure 7.7: Navigation menu (1).

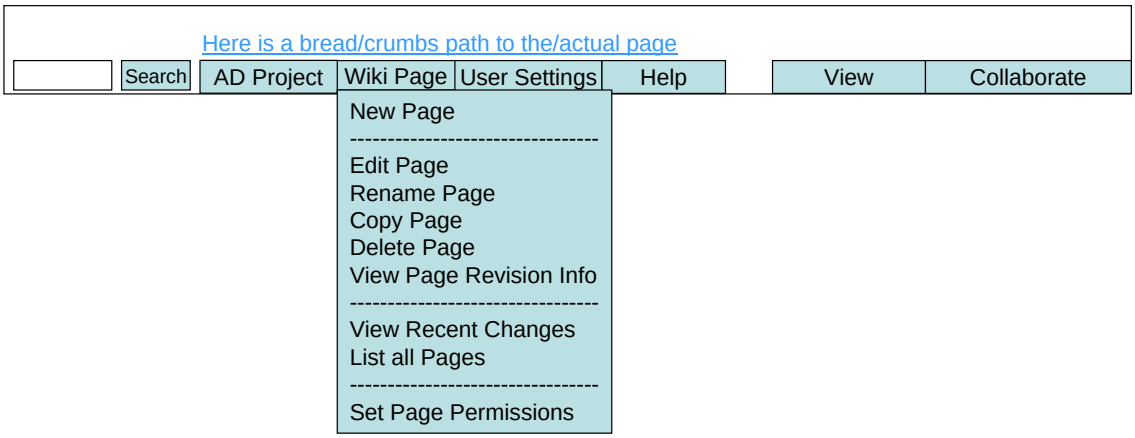


Figure 7.8: Navigation menu (2).

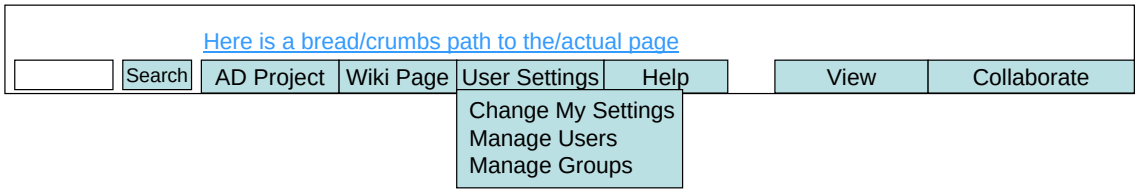


Figure 7.9: Navigation menu (3).

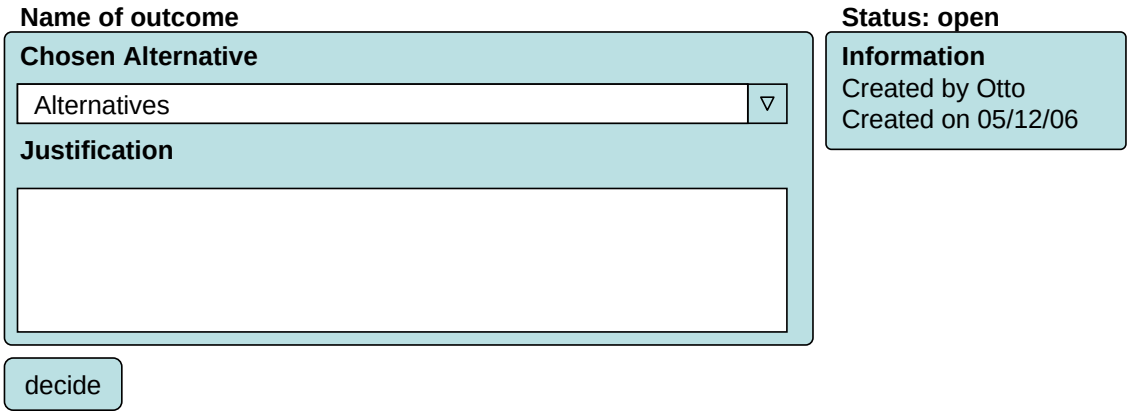


Figure 7.10: Open AD (e.g. if created automatically).

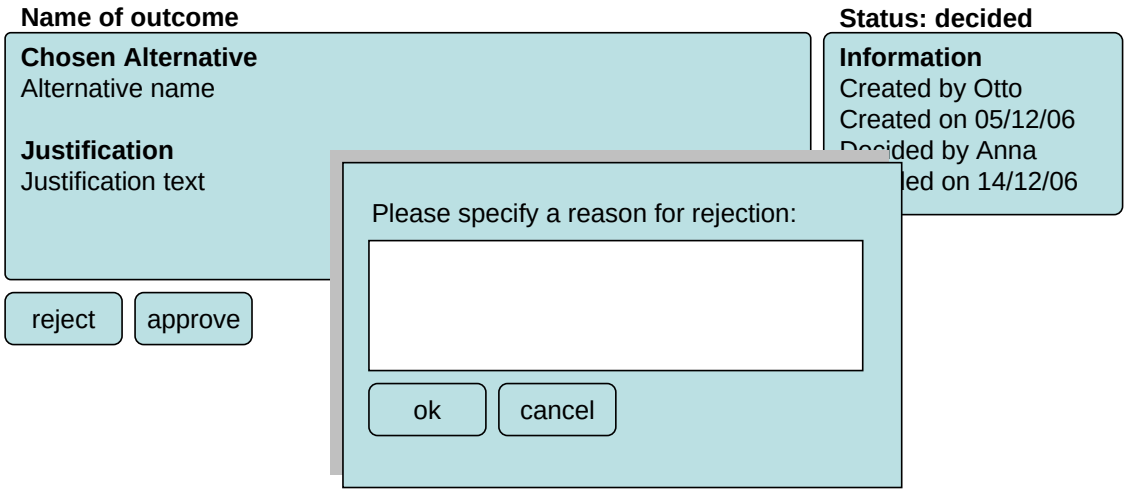


Figure 7.11: Decided AD, about to reject.

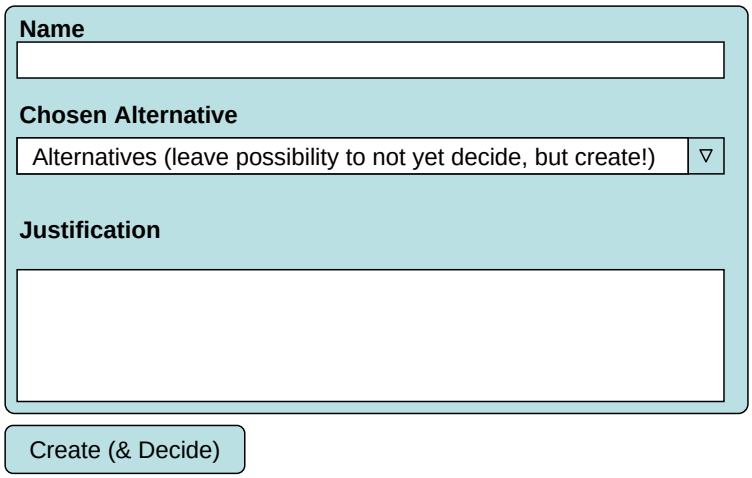
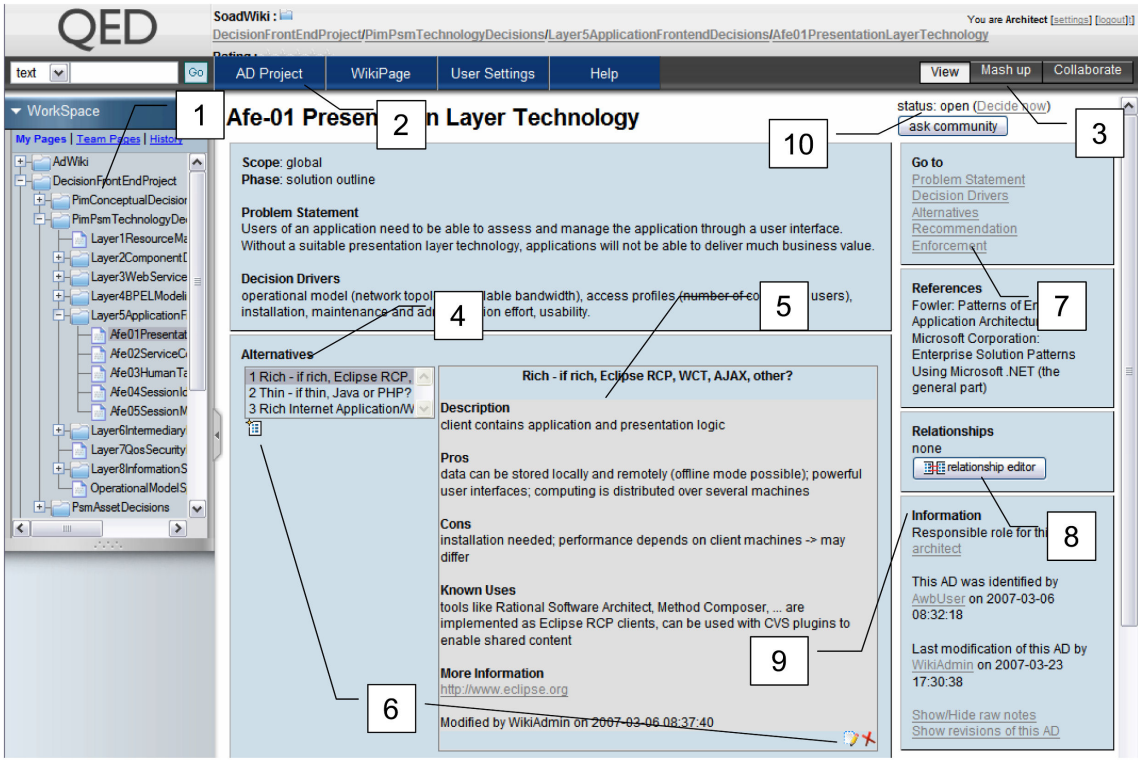
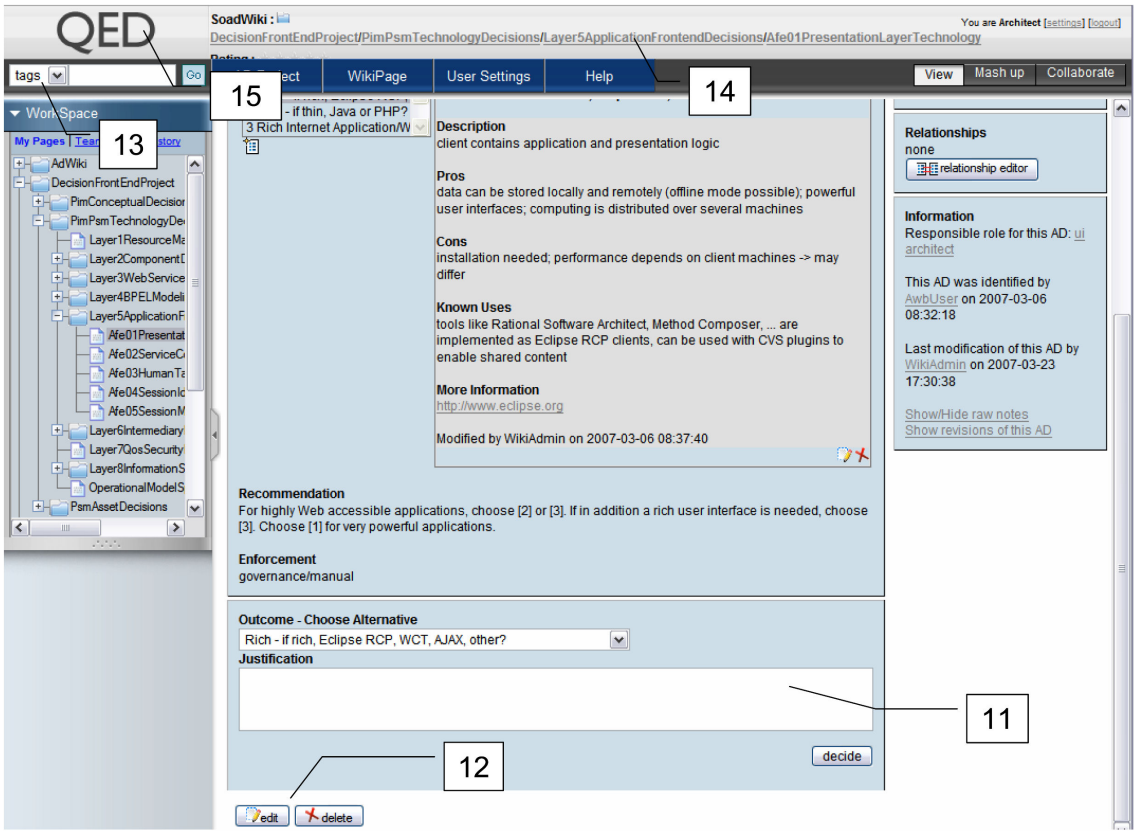


Figure 7.12: Create and decide AD.

Appendix D: AD_{kwik} Reference



(1) Content explorer (project and decision trees)
(2) Navigation menu
(3) Perspective switch (View - Mash up - Collaborate)
(4) Alternatives list (master)
(5) Alternatives description (details)
(6) Create, edit and delete AD alternative
(7) Quick links to AD attributes
(8) Relationship editor
(9) More information about the AD
(10) AD status



(11) Decision making box
(12) Edit or delete AD
(13) Tag and text search
(14) Full page path (breadcrumbs)
(15) Logo and link to start page

Appendix E: AD_{kwik} Components

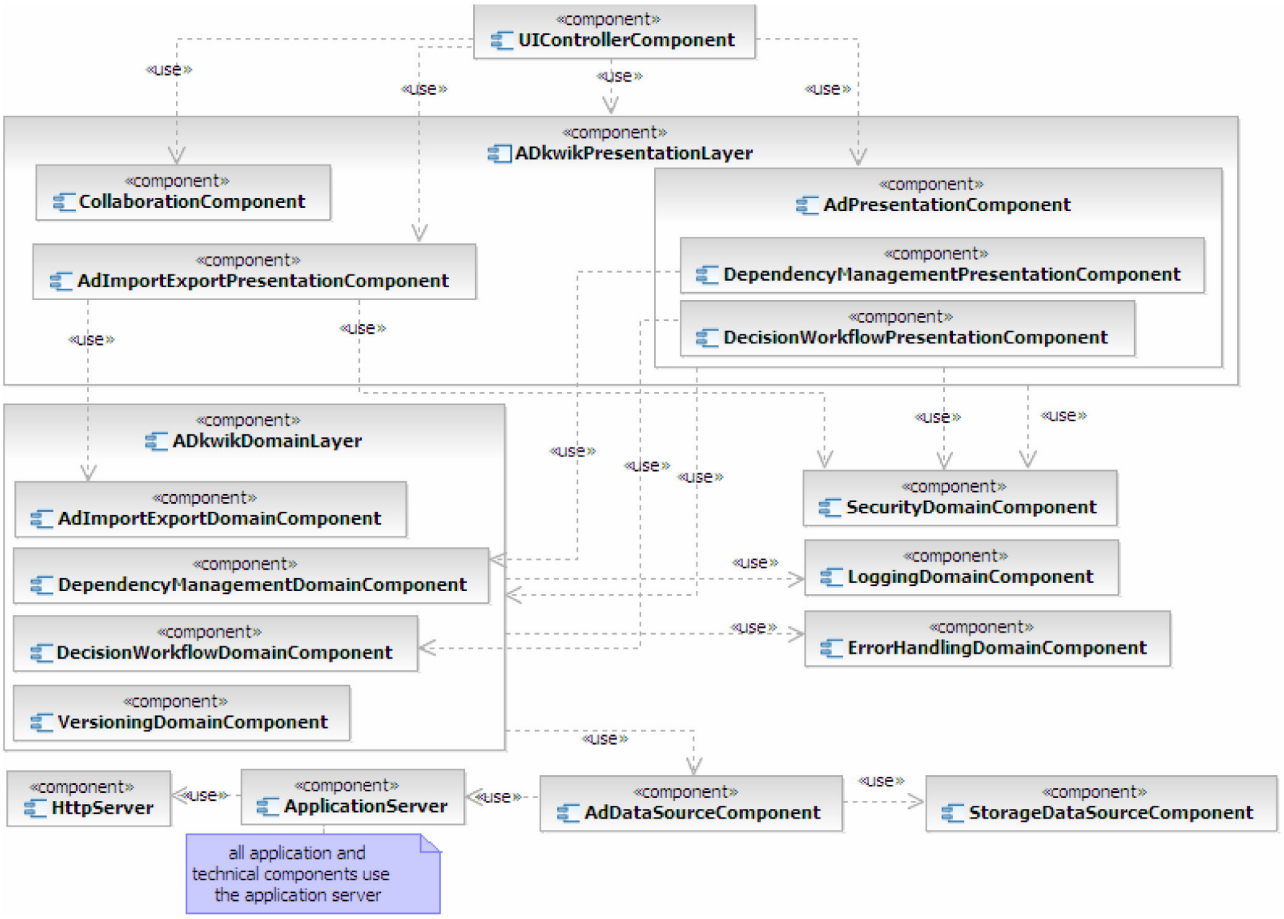


Figure 7.13: Components used in AD_{kwik} (including QEDWiki components).

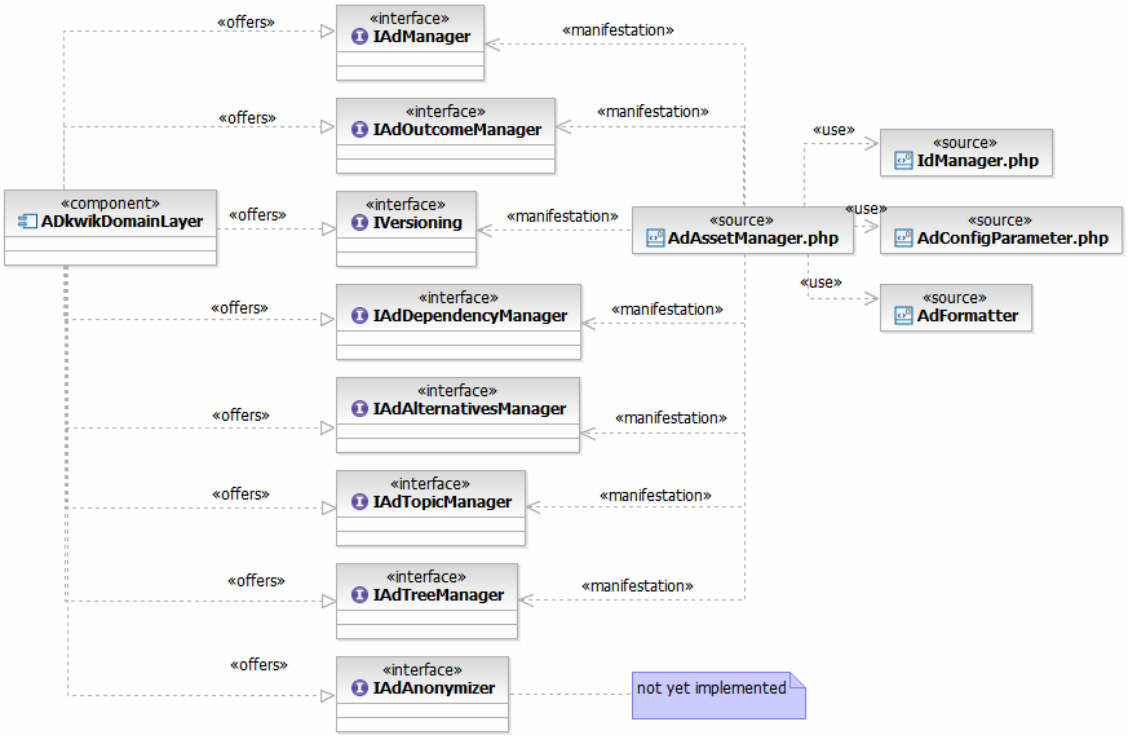


Figure 7.14: ADkwikDomainLayer.

Component ID and Name	**COMP-AD-01: ADkwikDomainLayer**
Responsibilities	1. create AD, AD topic, alternative, dependency, outcome 2. provide different attributes of an AD, AD topic, alternative, dependency, outcome 3. edit attributes of AD, AD topic, alternative, dependency, outcome 4. delete AD, AD topic, alternative, dependency, outcome
NFRs	Good performance is a must, because this component is the base of the system (the part which is most often used).
Interfaces	«offers» IAdManager IAdOutcomeManager IAdTreeManager IAdDependencyManager IAdAlternativesManager IAdTopicManager IVersioning IAdAnonymizer «uses» ILogActivities IExceptionHandling
Design Rationale	The ADkwikDomainLayer provides business behavior. It is used by the AdPresentationLayer to do all the documentation of an AD, alternatives, dependencies, AD topics and outcomes on asset basis (this means identification & documentation, making and enforcement).
Artifacts	**AdUtilites/AdAssetManager/AdAssetManager.php** - implements all interfaces **AdUtilities/AdAssetManager/IdManager.php** - provides a simple ID management (utility class) **AdUtilities/AdConfigParameter.php** - includes some parameters for data configuration **AdUtilities/AdFormatter.php** - includes some helper functions for formatting before database inserts etc.
Implementation Approach	The AdAssetManager manages the entity classes Ad_decision, Ad_alternative, Ad_dependency, Ad_topic, Ad_outcome. All save actions on an asset are versioned. All transactions are logged (using the QEDWiki functionality). Exception handling via QEDWiki functionality.

Table 7.3: ADkwikDomainLayer

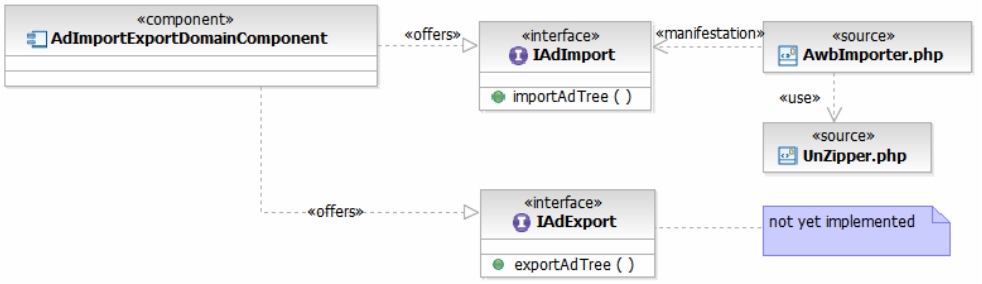


Figure 7.15: AdImportExportDomainComponent.

Component ID and Name	**COMP-AD-02: AdImportExportDomainComponent**
Responsibilities	1. import AD tree from outside the system and make it available 2. export AD tree 3. import single AD 4. export single AD
NFRs	Extensibility (not only AWB files, it must be possible to write other converters and include them into system)
Interfaces	«offers» IAdImport IAdExport «uses» ILogActivities IExceptionHandling All interfaces of ADkwikDomainLayer
Artifacts	**AdUtilities/AdImporter/AwbImporter.php** - inserts an awb file into database. **AdUtilities/AdImporter/UnZipper.php** - utility for the importer to unzip import file.
Implementation Approach	Parsing and converting AD content form file into respective AD_{kwik} object and storing it with the help of ADkwikDomainLayer interfaces.

Table 7.4: AdImportExportDomainComponent

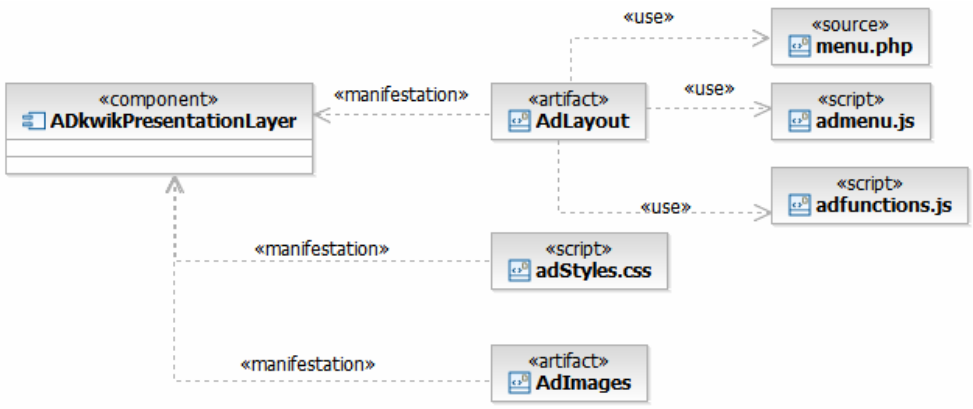


Figure 7.16: ADkwikPresentationLayer.

Component ID and Name	**COMP-UI-03: ADkwikPresentationLayer**
Responsibilities	1. provide overall wiki user interface
NFRs	Usability
Artifacts	**layouts/default/*** **layouts/default/admenu.js** - structure of the AD_{kwik} menu. **layouts/default/adfunctions.js** - some utility functions which are called when a menu entry of AD_{kwik} menu is selected. **images/*** - image files **other/soad/adstyles.css** - style sheets
Implementation Approach	Analogous to the default layout of QEDWiki.

Table 7.5: ADkwikPresentationLayer

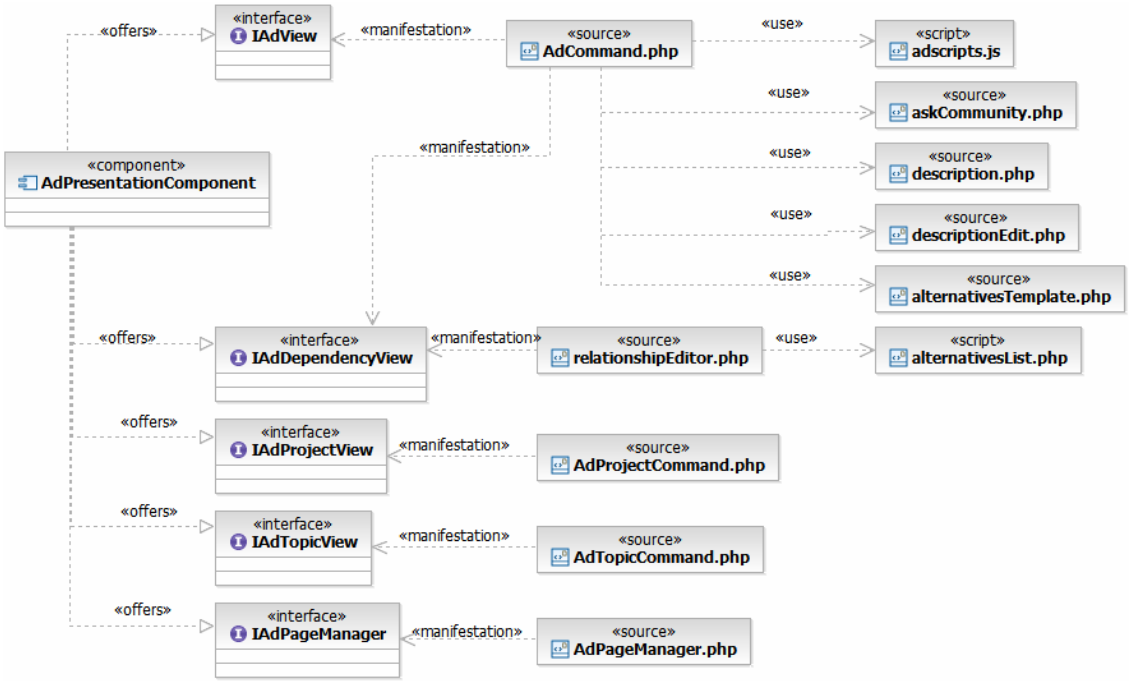


Figure 7.17: AdPresentationComponent.

Component ID and Name	**COMP-UI-04: AdPresentationComponent**
Responsibilities	1. provide view for (attributes of) AD, AD topic, alternative, dependency, outcome, AD project 2. provide view for editing different attributes of AD, AD topic, alternative, dependency, outcome, AD project 3. provide view for dependency management 4. provide view for state management of outcome
NFRs	Usability
Interfaces	«offers» IAdView IAdTopicView IAdProjectView IAdDependencyView IAdPageManager «uses» IAuthorisation All interfaces of ADkwikDomainLayer
Design Rationale	The AdPresentationComponent provides a user interface for the modification of ADs, AD topics, AD projects, dependencies, alternatives etc.
Artifacts	**Commands/AdCommand/latest/AdCommand.php** - provides a user interface for creating, editing, deleting an Ad, managing the state (outcome). Includes several artifacts (description.php, descriptionCreate.php or descriptionEdit.php) dependent on the status of the wiki and user input. **Commands/AdCommand/latest/adscripts.js** - some JavaScript scripts **Commands/AdCommand/latest/alternativesList.php** - combobox list of alternatives, needed in the relationship editor **Commands/AdCommand/latest/askCommunity.php** - template for "rating" how the different ADs were decided (which alternatives where chosen) **Commands/AdCommand/latest/relationshipEditor.php** - Relationship Editor, included in AdCommand.php **Commands/AdCommand/latest/alternativesTemplate.php** - shown in the right area when an alternative is selected. **Commands/AdCommand/latest/description.php** - the description of the AD **Commands/AdCommand/latest/descriptionCreate.php** - the form which is shown when user creates a new AD **Commands/AdCommand/latest/descriptionEdit.php** - the form which is showed when user edits an existing AD **Commands/AdProjectCommand/AdProjectCommand.php** - provides user interface for creating an AD project **Commands/AdTopicCommand/AdTopicCommand.php** - provides user interface for creating and editing an AD topic **AdUtilities/AdPageManager.php** - creates different types of pages for the QEDWiki tree dependent on the user input. These pages contain different commands (AD, AD topic or AD project).
Implementation Approach	This component is split into several QEDWiki plug-ins (commands) which are called by the QEDWiki. This means it is not application independent! This component uses ADkwikDomainLayer to transform user input. It incorporates the DependencyManagementPresentationComponent and the DecisionWorkflowPresentationComponent.

Table 7.6: AdPresentationComponent

Figure 7.18: AdImportExportPresentationComponent.

Component ID and Name	**COMP-UI-05: AdImportExportPresentationComponent**
Responsibilities	1. provide user interface for importing AD tree from outside the system and make it available 2. provide user interface for exporting AD tree 3. provide user interface for importing single AD 4. provide user interfaces for exporting single AD
NFRs	Usability
Interfaces	«offers» IAdImportView IAdExportView «uses» All interfaces of AdImportExportDomainComponent
Artifacts	**Commands/AwbImporter/latest/AwbImporter.php** - provides user interface for importing an .awb file
Implementation	This component is implemented as a QEDWiki command. It uses the AdImportExportDomainComponent for storing. IExportView is not yet implemented.

Table 7.7: AdImportExportPresentationComponent

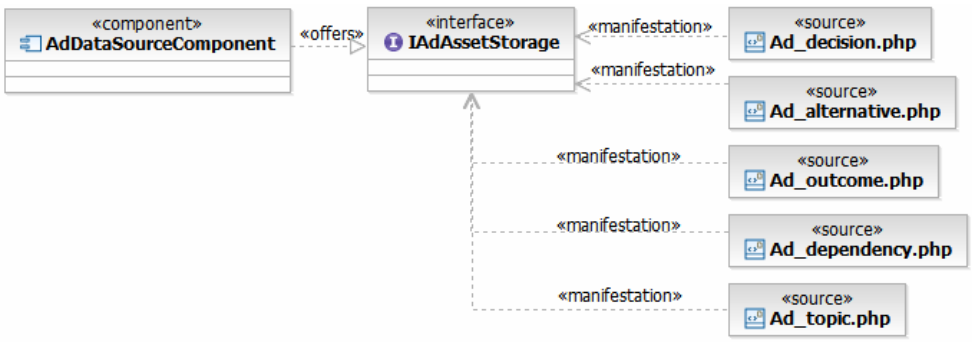


Figure 7.19: AdDataSourceComponent.

Component ID and Name	**COMP-DATA-06: AdDataSourceComponent**
Responsibilities	1. provide simplified access to storage
NFRs	Acid, transaction
Artifacts	**AdUtilities/AdEntities/Ad_*.php** - the used entities
Implementation	Provided by the Zend Framework (ZActiveRecord Pattern). Database tables are generated automatically.

Table 7.8: AdDataSourceComponent

Appendix F: AD_{kwik} Decisions

Figure 7.20: AD ‘Method Selection’ on the concept level.

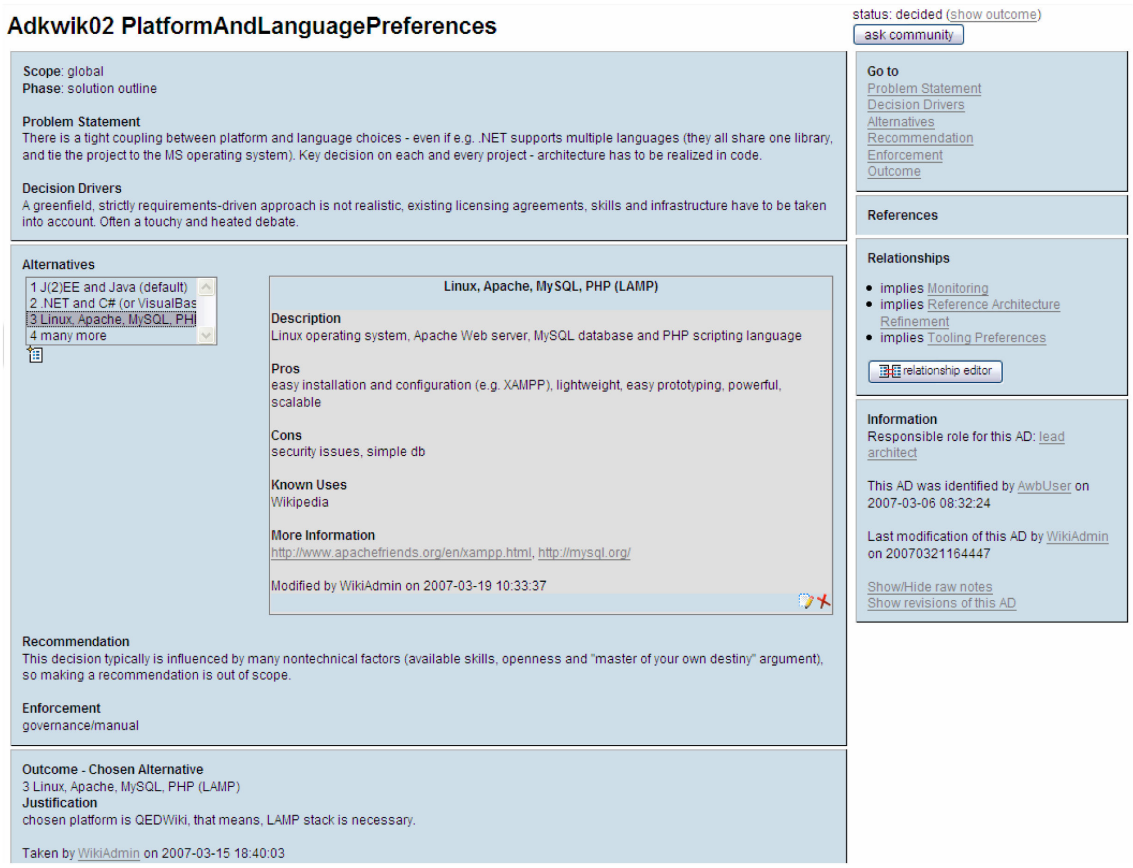


Figure 7.21: AD 'Platform and Language Preferences' on the concept level.

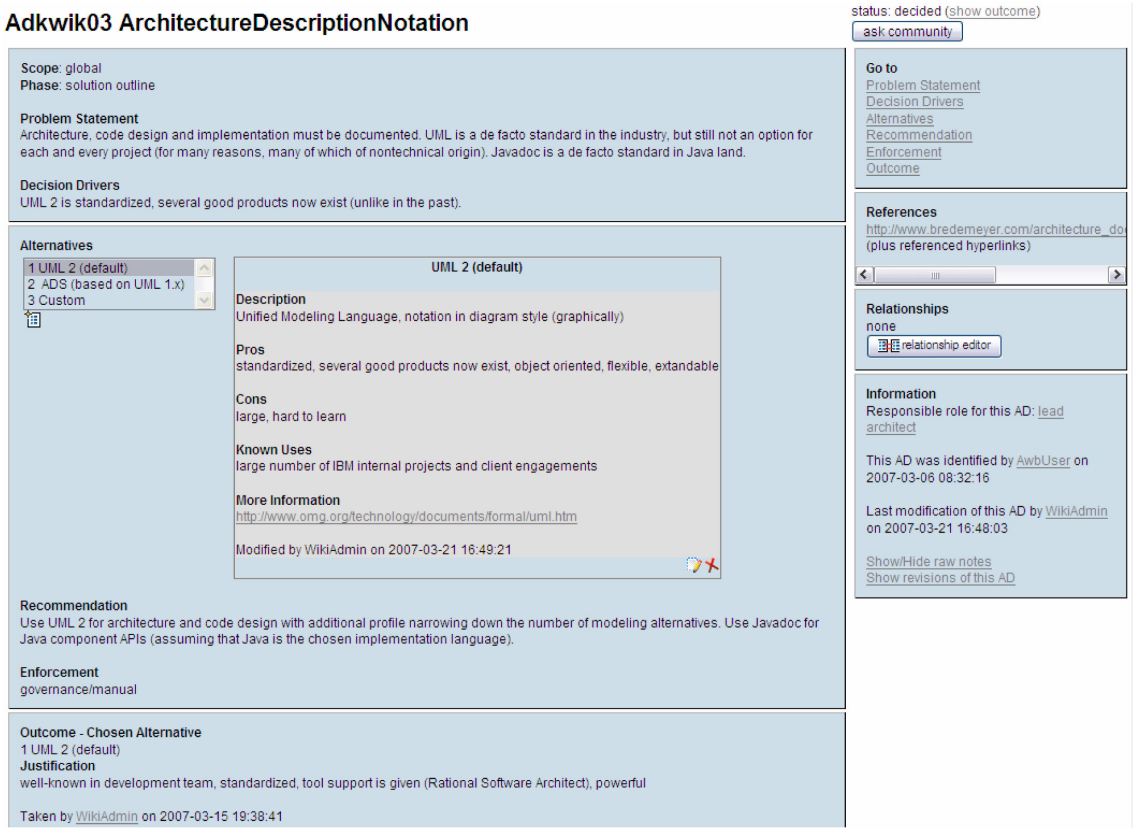


Figure 7.22: AD 'Architecture Description Notion' on the concept level.

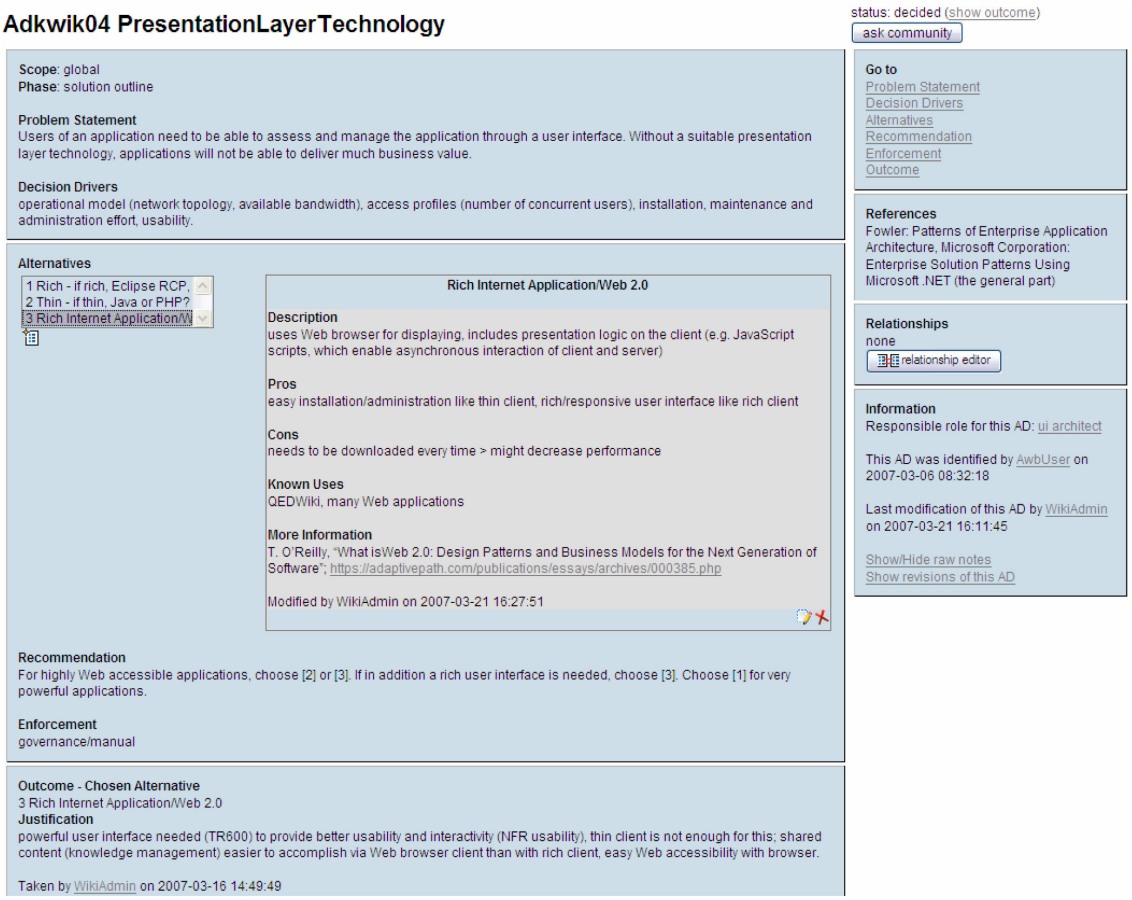


Figure 7.23: AD 'Presentation Layer on the technology level.

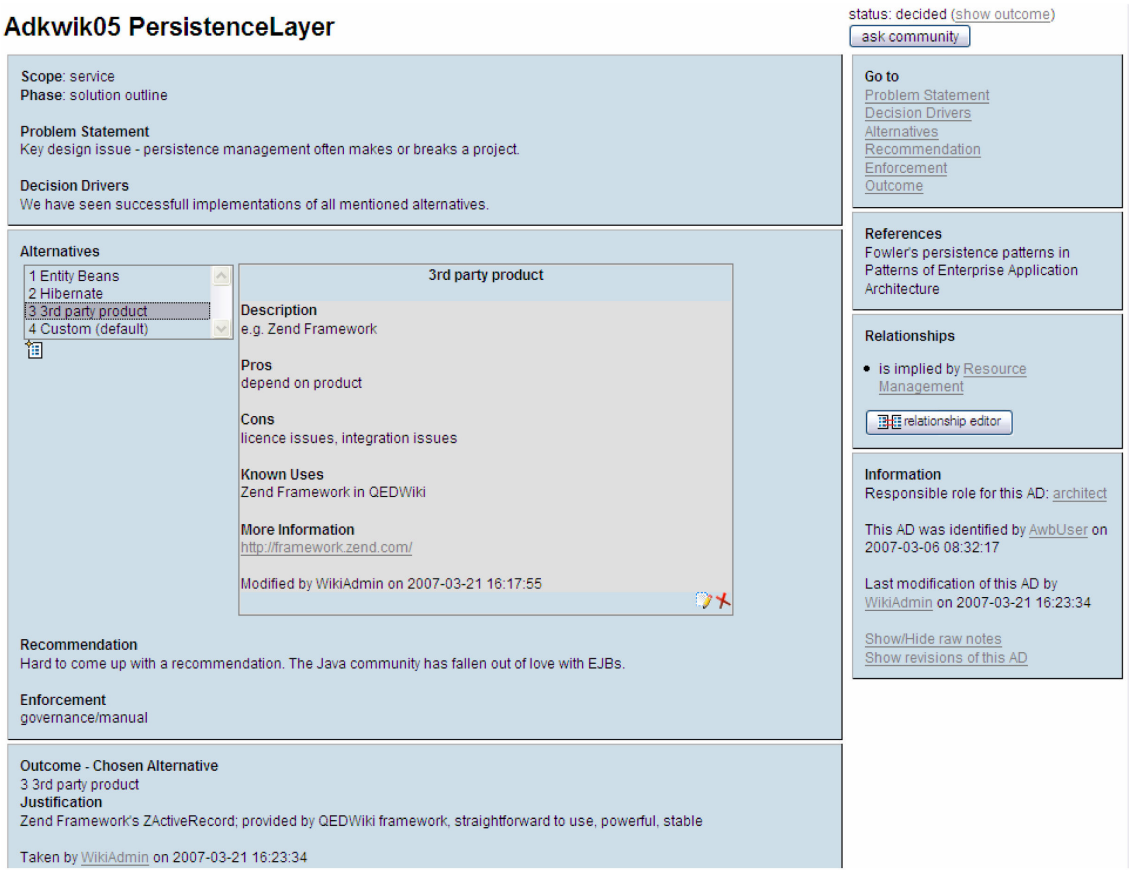


Figure 7.24: AD 'Persistence Layer' on the asset level.

Appendix G: First User Tests

First User Test December 2006

Usability Test with Olaf Zimmermann - Report
December 20, 2006 – very early state of prototype, test focused on user interface and terminology

Question	Expected answer	Answer	Result
What do you think does the menu entry "Page" mean?	everything related to AD content	The TP (test person) assumes something to "get content".	
What do you think does the menu entry "View" mean?	everything related to wiki content	"the same..."	The menu is not intuitive to use. Suggestion by the TP: use domain specific terms (e.g. "Decision Page" instead of "Page"), or menu entries for decision platform and for wiki platform. The wiki can totally be adapted to the domain. Page will be renamed into "AD Project", View into "Wiki Page" (to be tested with a larger community).
You are Project Lead Architect of a new SOA project and want to set up the wiki for your project team. How do you do this?	log in, import a decision tree (page -> import ADs, fill out input form), create users	This question could only be solved with the help of the moderator. The TP expected something in the tree or page > New Project. With some help, the import function was finally found.	TP suggested to provide a start page with information how to set up the wiki for use in an architect project. The AdWiki entry (which is described in the FAQs) in the tree is confusing as it is not clear what it means. "Is it from other project teams? Can I delete it?" "Utils" is something nice to have (the term is misleading), will be renamed. Button "send" will be renamed to "start import".
The wiki was set up successfully. Your team started to work with it. You discovered a new transaction management pattern and want to document it.	Selection of the AD, creating a new alternative	This question was solved without problems.	The user interface for ADs and alternatives will not be changed.

The team discusses on the new pattern. You want to motivate it and comment your solution in the wiki. Moreover, you add a new paper, which explains the pattern in detail.	"Share" > comment, add attachment	The "Share" button was not found. Even as it was found, the TP could not understand it because of the naming.	TP suggested names like "team collaboration", "forum", "decision communication" and so on. The location is ok, once known you will find it whenever you need it. Location will not be changed; "Share" will be renamed to "Collaborate".
You decide to use this solution in the new SOA system. How to you communicate this to the wiki?	selection of AD, click button decide on the bottom	This could not be tested, because all imported decisions already were decided.	There must be a possibility to select the alternative directly (not copy and paste...). Click button "decide" on bottom or directly select alternative is ok. Import needs to be adopted: outcomes will not be imported.
Your decision directly influences other decisions. For example, the decision about transactionality depends on its outcome. How do you communicate this to the system?	selection of AD, edit, add relationship	This question could only be solved with the help of the moderator. The TP assumes the edit possibility directly in the relationship box.	A relationship editor will be implemented, which can be reached through the relationship box.
The project is finished. All decisions are documented, now the wiki needs to be cleaned up. First of all, you want to replace all customer names with fictitious names. How do you perform this?	page -> search and replace	This question could only be solved with the help of the moderator. The TP assumes right click on the tree root and press anonymize.	There can be something additional in the menu. TBD. This feature will not be part of the prototype.
You are almost done! :) Now you can export your decision tree and delete it.	page -> export page -> delete decision	Was clear.	No changes.

Table 7.9: First user test.

Second User Test March 2007

The following text is taken and adapted from notes of the test person and by courtesy of Olaf Zimmermann.

Usability and function test of AD_{kwik} in comparison with MS Word Arc-100 tables and AWB as configured/enhanced in SOAD project in 2006
March 09, 2007 – stable prototype, key use cases implemented

Tester:
Olaf Zimmermann, IT architect: deep experience with Arc-100 (key part of day job 1999-2005), medium experience with AWB (since Q2006), early AD_{kwik} adopter (since December 2006)

Test data:
57 executive, conceptual integration, and technical Web services decisions as required on Sparkassen Informatik SOA project described in http://soadecisions.org/download/pra06-zimmermann.pdf (pre-populated from reusable SOAD asset structured according to meta model described in [10])

Test objects:
a) Arc-100 template in MS Word 2002 (created by AWB 0.4.57, identical to the one described in IGSM WPD Version 4.1.1., August 2002)
b) AWB version 0.4.57 from TJW (installed mid 2006 from latest version in AWB TR)
c) AD_{kwik} alpha driver from Feb 2007, based on QED Wiki version 1.0.0 for Windows

Test cases:
0: Import and export capabilities
1: Search and browse ADs and AAs
2: Understand AD/AA details
3a: Modify AD/AA content
3b: Refactor ADs and AAs: rename, regroup
4: Add and categorize AD/AA
5: Discuss AD/AAs with team and peers in the community
6: Make and document decision
7: Enforce decision (communicate and control)

Scores:
(1) Excellent/outstanding (2) Good (3) Ok (4) Acceptable (5) Poor (6) Not existing/unusable.

Test case	**Arc-100 in MS Word**	**AWB**	**AD_{kwik}**
0: Import and export capabilities	Manual effort, various formats, text only (4)	HTML Arc-100, zip file, CVS (3)	Import from AWB, export to Arc-100 planned but not yet implemented, Wiki pages, RDBMS (2) for concept, (3) for current version
1: Search and browse ADs and AAs	TOC, text search, sequential document, relationship management is manual task (2) for up to 100 ADs (5) for large knowledge base	Explorer, but no text search, XML model, configurable e.g. via reminders view, powerful relationship management (but single type only by default, AD level only) (2)	Explorer, text and tag search, list all pages, master details, breadcrumbs, extensible via domain model/RDBMS queries, powerful relationship management (multiple types, both AD and AA level) (1)
2: Understand AD/AA details	(3) for small amount of information (4) for large AD/AA description	Several tabs have to be looked at, background information and outcome capturing not separated (4)	All relevant information appears on first screen, outcome factored out. AD/AA fields might be too small for novice user, who will have to read attachments and follow references frequently (same issue in AWB and Arc-100) (1) for experienced architect (4) for newcomer
3a: Modify AD/AA content	Copy and paste, WYSIWYG (3) for small modifications (4) for larger edits	Copy and paste, WYSIWYG (2) for small modifications (3) for larger edits	Copy and paste, separate window (2) for small modifications (3) for larger edits
3b: Refactor ADs and AAs: rename, regroup	(5) for small documents (6) for large knowledge base	(1) drag and drop in explorer, all changes automatically propagated throughout model, powerful Eclipse tooling	(4) WikiName renaming via typing required for regrouping, renaming is easier

4: Add and categorize AD/AA	No support for categorization and relationship management other than TOC, copy and paste (3) for up to 70 ADs (6) for large knowledge base	Via Prototypes/Palette, no copy and paste (2) if AAs are short alternatives (4) if AAs are separate model elements	categorization is easy via drag and drop Via separate window, no copy and paste of entire element (?). Powerful relationship editor. Categorization tedious. (3) for small number (4) for huge knowledge base
5: Discuss AD/AAs with team and peers in the community	(4) the comment feature in word and emails with attachments are the only support (I appreciate the spell checker though)	There is a role concept, but working in a team on one AWB project is painful (XML files in CVS) (6)	Standard wiki features can be used (comments, email, etc.), plus versioning and ask community button (2)
6: Make and document decision	Separate text field, not aligned with AAs Needed one minute per decision in test (scrolling), number to be verified, could be high because this was the first test (3)	Separate node attribute, not aligned with AAs. Needed ten seconds per decision in test, number to be verified (3)	AA can be selected from menu, so only effort is to type justification (which can not be edited later). Needed one second per decision in test, number to be verified (2)
7: Enforce decision	(6) no support except for "RDFD"	(5) resolved flag and reminders, but no real support for decision tracking once resolved	Decision life cycle workflow, RDBMS reports, user management (2)

Table 7.10: User test in March 2007.

Questions on the information architecture/terminology
How do you like the navigation menu (top)?
The menu is fine, in the beginning it was a bit tough for me to understand what is provided by AD_{kwik} and what is provided by QEDWiki. At that time, there was no users guide.
What is good, was is bad?
I appreciate that there are less than 10 options per menu. The Eclipse designers should take a look :) There might be *some* room for improvements in the grouping – e.g. why is "View Revision Info" where "Rename" and "Delete AD" are? Why no modify there?
How do you get along with the terms? Obscurities or terms in the menu which you did expect to be located somewhere else?
The naming of UI and model elements is fine with me, I have been using terms like AD and AA for years. The AWB Import menu needs work, though: when reloading a master zip file, I am informed about an already existing tree rather late. And I have to select the zip file twice, and define the project it is supposed to go to (which is already known from my navigation).

Questions on the AD page
How is the adjustment of the "boxes" on the AD page? What is good, what is bad?
I like that all background information is visible when I open an AD. Order is fine, before making a decision I first want to understand what the AD is about, what requirements led to it (DDs), what my alternatives are. Next I am interested in detailed reasoning (pros, cons, recommendations), dependencies and more information. The Outcome will have to go to a separate page, as there is a 1:n relationship between AD and AdOutcome e.g. needed for the decision scoping concept in SOAD.
Edit/delete/add buttons for AD and alternatives - how is the adjustment? If bad, how did you expect it to be?
Ok in general, sometimes I struggle to find the edit AD button (bottom of the page). Maybe it would be better to move it up?

Questions on the period of orientation
Long/short/average?
When started working with the system, I could immediately relate to the concepts and found everything in the details views. The menus needed some time to grasp.

Questions on performance
How are the response times? Which response times are too long?
In single user mode and on my local T60 I constantly experience sub second response times in test cases 1-9. I stress tested the database with several thousand ADs in more than 10 projects, no performance degradation occurred. Our test user in Berlin accessed the same AD_{kwik} instance on my machine in ZRL, and did not complain about lack of responsiveness or so. Scalability tests with 10-20 concurrent users should be run next. The AWB import is rather slow, 20 seconds for 100 decisions. This should be improved. There should be a create-from-master-in-database feature.

Things that you like?
Fonts, colors, the entire user experience. And architecture and vision of course ;)

Things that bother you?
Nothing comes to my mind right now :)