# Building Up and Reasoning About Architectural Knowledge

Philippe Kruchten[1], Patricia Lago[2], and Hans van Vliet[2]

[1] University of British Columbia
Vancouver, Canada
pbk@ece.ubc.ca
[2] Vrije Universiteit
Amsterdam, the Netherlands
{patricia, hans}@few.vu.nl

**Abstract.** Architectural knowledge consists of architecture design as well as the design decisions, assumptions, context, and other factors that together determine why a particular solution is the way it is. Except for the architecture design part, most of the architectural knowledge usually remains hidden, tacit in the heads of the architects. We conjecture that an explicit representation of architectural knowledge is helpful for building and evolving quality systems. If we had a repository of architectural knowledge for a system, what would it ideally contain, how would we build it, and exploit it in practice? In this paper we describe a use-case model for an architectural knowledge base, together with its underlying ontology. We present a small case study in which we model available architectural knowledge in a commercial tool, the Aduna Cluster Map Viewer, which is aimed at ontology-based visualization. Putting together ontologies, use cases and tool support, we are able to reason about which types of architecting tasks can be supported, and how this can be done.

## 1 Introduction

Software that is being used, evolves. For that reason, quality issues like comprehensibility, integrity, and flexibility are important concerns. For that reason also, we not only bother about today's requirements during development but also, and maybe even more so, about the requirements of tomorrow.

This is one of the main reasons for the importance of software architecture, as for instance stated in Bass *et al.* [1]: a software architecture manifests the early design decisions. These early decisions determine the system's development, deployment, and evolution. It is the earliest point at which these decisions can be assessed.

There are many definitions of software architecture. Many talk about components and connectors, or the 'high-level conception of a system'. This high-level conception then is supposed to capture the 'major design decisions'. Whether a design decision is major or not really can only be ascertained with hindsight, when we try to change the system. Only then it will show which decisions were really important. A priori, it is

C. Hofmeister et al. (Eds.): QoSA 2006, LNCS 4214, pp. 43 – 58, 2006.
© Springer-Verlag Berlin Heidelberg 2006

often not at all clear if and why one design decision is more important than another one [9].

Architectural design, even well documented according to all the good recipes [5, 12, 14], is only one small part of the *Architectural Knowledge* that is required to design a system, or that is needed to guide a possibly multisite development team, or that can be exploited out of a system to build the next one, or that is required to successfully evolve a system. Van Vliet and Lago have pointed rightfully that all the assumptions that were made during the architectural design, all the linkage to the environment are a key component of architectural knowledge [16, 28]. Similarly, Bosch and others have pointed out that design decisions, the tight set of interdependencies between them, and their mapping to both the requirements, needs, constraints upstream, or the design and implementation downstream are also a key component of architectural knowledge [2, 15, 25].

We can usually get at the architectural *Design* part, ultimately by reverse engineering if there was no explicit documentation. This amounts to the *result* of the design decisions, the solutions chosen, not the reasoning behind them. The *Context* and some of the *Rationale* may be partially retrieved from management documents, vision documents, requirements specs, etc. Design *Decisions* and much of the *Rationale* are usually lost forever, or reside only in the head of the few people associated with them, if they are still around.

So the reasoning behind a design decision, and other forces that drive those decisions (such as: company policies, standards that have to be used, earlier experiences of the architect, etc.), are not explicitly captured. This is tacit knowledge, essential for the solution chosen, but not documented. At a later stage, it then becomes difficult to trace the reasons of certain design decisions. In particular, during the evolution one may stumble upon these design decisions, try to undo them or work around them, and get into trouble when this turns out to be very costly if not impossible. The future evolutionary capabilities of a system can be better assessed if this type of knowledge would be explicit. We use the term *assumptions* as a general denominator for the forces that drive architectural design decisions. Just like it is difficult to distinguish between the *what* and the *how* in software development, so that one person's requirements is another person's design, it is also difficult to distinguish between assumptions and decisions. Here too, from one perspective or stakeholder, we may denote something as an assumption, while that same thing may be seen as a design decision from another perspective. As a result, we are left with:

**Architectural Knowledge = Design Decisions + Design** (1)

In this paper, we focus on the Design Decisions and their rationale. We distinguish four types of design decisions:

- **Implicit and undocumented:** the architect is unaware of the decision, or it concerns "of course" knowledge. Examples include earlier experience, implicit company policies to use certain approaches, standards, and the like.
- **Explicit but undocumented:** the architect takes a decision for a very specific reason (e.g., the decision to use a certain user-interface policy because of time constraints). The reasoning is not documented, and thus is likely to vaporize over time.

- **Explicit, and explicitly undocumented:** the reasoning is hidden. There may be tactical company reasons to do so, or the architect may have personal reasons (e.g., to protect his position).
- **Explicit and documented:** this is the preferred, but quite likely exceptional, situation.

### 1.1 The Role of Knowledge Management

The main value of a software company is its intellectual capital. As Rus and Lindvall [21] state: *The major problem with intellectual capital is that it has legs and walks home every day*. This is not only a problem when a key person, such as a software architect, goes on holiday, moves on to the next project, or even quits his job, but also when the company educates staff. It is in the interest of companies to transform architectural knowledge, such as design decisions, from the architects' minds to explicit knowledge on paper. Individual experts should share their knowledge amongst each other and with the rest of the company. The field of research that studies these topics is called *knowledge management*. A key motivation for our research is to support the sharing of architectural knowledge.

### 1.2 Dimensions of Architectural Knowledge

Nonaka and Takeuchi identify three levels of knowledge [20]:

- **Tacit:** mostly in the head of people
- **Documented:** there is some trace somewhere
- **Formalized:** not only documented, but organized in a systematic way.

The same categorization may be applied to architectural knowledge. The first three types of design decisions identified above then are examples of tacit knowledge. We aim to formalize part of this tacit knowledge. We readily recognize that only part of this knowledge can, and need, be formalized. We need only formalize what is subsequently useful to persons that exploit the architectural knowledge. To get insight into this need, we developed a use-case model for architectural knowledge.

As shown in figure 1, there is also a level of maturity of architectural knowledge: some design decisions, or elements of the design may be tentative, not fully integrated, whereas others are hard coded, immutable elements.

Finally, there is a time dimension to architectural knowledge. Certain architectural knowledge may be valid or relevant in some version of the architecture and/or system, but might be overridden, become invalid or irrelevant after a certain modification is made. Thus, we should not only retain the latest version of the architectural knowledge, but its version history as well.

### 1.3 Design Rationale

Capturing design rationale has been a key research topic for many years, leading to interesting models, tools and methods [4, 6, 13, 18, 19], but it has failed to transfer to

practice [3, 17]. Why? This is mostly because the burden to capture assumptions and decisions outweighs largely the immediate benefits that the architect may draw. These benefits would be felt much later, or by others. If we are not careful to address the key problem: how to move this knowledge out of the tacit level into at least the documented level and then the formalized level, all what we may do with architectural knowledge could follow the same route as design rationale has done over the years: nice ideas, but not practical. One way is to automate the collection of rationale (or of decisions, or both). These observations are corroborated in a recent empirical study of architecture design rationale: documenting architecture design rationale is deemed important, but methodology and tool support is lacking [22].

### 1.4 Contribution of the Paper

The remainder of this paper is devoted to a discussion of what Architectural Knowledge entails, in terms of an ontology for design decisions, and typical usages thereof (with a focus on design decisions), followed by a sketch of the extent to which a commercial tool, the Aduna Cluster Map Viewer, supports the storage and use of Architectural Knowledge. This then leads to an agenda of research questions we think need answers for Architectural Knowledge modeling and usage to become a practical reality. These research questions mainly concern suitable visualization and task-specific support. As a running example, we use the set of design decisions of the SPAR Aerospace Robotic Arm. A companion paper [26] gives a more elaborate discussion of the use-case model, including a sample application of some of these use cases in an industrial application.

## 2 An Ontology of Design Decisions

In this section we describe an ontology of architectural design decisions, and their relationships. An earlier version hereof was published in [15]. This ontology will later be used to structure architectural knowledge of the SPAR Aerospace Robotic Arm. The use cases of section 3 refer to this structure, and tools like the Aduna Cluster Map Viewer operate on architectural knowledge structured this way.

### 2.1 Kinds of Architectural Design Decisions

#### 2.1.1 Existence Decisions (“Ontocrises”)

An existence decision states that some element/artifact will positively show up, i.e., will *exist* in the system’s design or implementation.

There are *structural decisions* and *behavioral decisions*. Structural decisions lead to the creation of subsystems, layers, partitions, components in some view of the architecture. Behavioral decisions are more related to how the elements interact together to provide functionality or to satisfy some non functional requirement (quality attribute), or connectors. *Examples:*

– Dexterous Robot (DR) shall have a Laser Camera System.
– DR shall use the Electromagnetic (EM) communication system to communicate with GroundControl.

Existence decisions are not in themselves that important to capture, since they are the most visible element in the system's design or implementation, and the rationale can be easily captured in the documentation of the corresponding artifact or element. But we must capture them to be able to relate them to other, more subtle decisions, in particular alternatives (see section 2.3).

### 2.1.2 Bans or Non-existence Decisions ("Anticrises")

This is the opposite of an existence decision, stating that some element will *not* appear in the design or implementation. They are a subclass of existential decisions in a way.

This is important to document precisely because such decisions are lacking any "hooks" in traditional architecture documentation. They are not traceable to any artifact present. Ban decisions are often made as we gradually eliminate possible alternatives. *Examples:*

– DR shall not block HST solar arrays, or communications systems.

### 2.1.3 Property Decisions ("Diacrises")

A property decision states an enduring, overarching trait or quality of the system. Property decisions can be design rules or guidelines (when expressed positively) or design constraints (when expressed negatively), as some trait that the system will not exhibit. Properties are harder to trace to specific elements of the design or the implementation because they are often cross-cutting concerns, or they affect too many elements. Although they may be documented in some methodologies or process in Design guidelines (see RUP, for example), in many cases they are implicit and rapidly forgotten, and further design decisions are made that are not traced to properties. *Examples:*

– DR motion should be accurate to within +1 degree and +1 inch.
– DR shall withstand all loads due to launch.

### 2.1.4 Executive Decisions ("Pericrises")

These are the decisions that do not relate directly to the design elements or their qualities, but are driven more by the business environment (financial), and affect the development process (methodological), the people (education and training), the organization, and to a large extent the choices of technologies and tools. Executive decisions usually frame or constrain existence and property decisions. *Examples:*

- Process decisions:

– All changes in subsystem exported interfaces (APIs) must be approved by the CCB (Change Control Board) and the architecture team.

- Technology decisions:

– The system is developed using J2EE.
– The system is developed in Java.

- Tool decisions:

– The system is developed using the System Architect Workbench.

Software/system architecture encompasses far more than just views and quality attributes *à la* IEEE std 1471-2000 [13]. There are all the political, personal, cultural, financial, technological aspects that impose huge constraints, and all the associated decisions are often never captured or they only appear in documents not usually associated with software architecture.

## 2.2 Attributes of Architectural Design Decisions

This subsection contains a list of attributes we deem essential. It may be extended with other attributes, such as cost, or risks associated with the design decision.

### 2.2.1 Epitome (or the Decision Itself)

This is a short textual statement of the design decision, a few words or a one-liner. This text serves to summarize the decisions, to list them, to label them in diagrams.

### 2.2.2 Rationale

This is a textual explanation of the "why" of the decision, its justification. It should not simply paraphrase or repeat information captured in other attributes, but have some valued added. If the rationale is expressed in a complete external document, for example, a tradeoff analysis, then the rationale points to this document. Note that rationale has two facets: an intrinsic rationale as a property of the design decision, and an extrinsic one, represented by its relationships to other design decisions. The latter is contained in any of the relationships discussed in section 2.3.

### 2.2.3 Scope

Some decision may have limited scope, in time, in the organization or in the design and implementation (see the Overrides relationship below). By default (if not documented) the decision is universal. *Examples:*

- **System scope:** The Communication subsystem [is coded in C++ and not in Java]
- **Time scope:** Until the first customer release [testing is done with Glider].
- **Organization scope:** The Japanese team [uses a different bug tracking system]

### 2.2.4 Author, Time-Stamp, History

The person who made the decision, and when the decision was taken. Ideally we collect the history of changes to a design decision. Important are the changes of State, of course, but also changes in formulation, in scope, especially when we run incremental architectural reviews. *Example:*

– "Use the UNAS Middleware"—tentative (Ph. Kruchten, 1993-06-04); decided (Ph. Kruchten, 1993-08-05); approved, (CCB, 1994-01-16); Scope: not for test harnesses; (Jack Bell, 1994-02-01); approved (CCB, 1994-02-27).

### 2.2.5 State

Like problem reports or code, design decisions evolve in a manner that may be described by a state machine or a statechart. See fig.1. This scheme may be too simple for certain environments, or too complicated for others; it has to match a specific decision and approval process. The states can be used to make queries, and as a filter when visualizing a Decision Graph; for example, omit ideas, or display them in green.

You would not include the ideas, tentative, and obsolesced decisions in a formal review, for example.

There is an implied “promotion” policy, which is used to check consistency of decision graphs (models), with the level of state being successively: 0: idea and obsolesced; 1: rejected; 2: tentative and challenged; 3: decided; 4: approved.

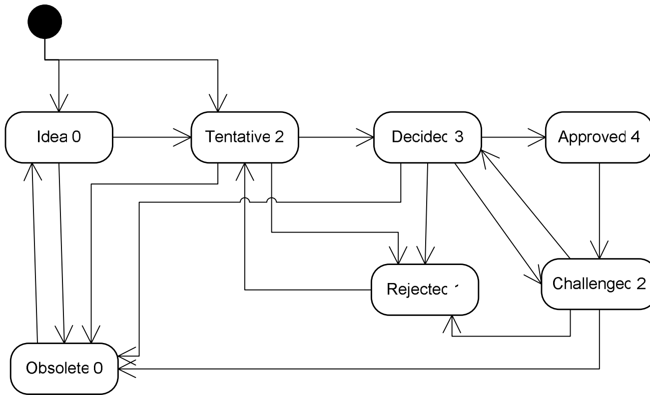


**Fig. 1.** State machine for a decision. **Idea**: Just an idea, captured not to be lost, when doing brainstorming, looking at other systems etc.; it cannot constrain other decisions other than ideas. **Tentative**: allows running “what if” scenarios, when playing with ideas. **Decided**: current position of the architect, or architecture team; must be consistent with other, related decisions. **Approved**: by a review, or a board (not significantly different from decided, though, in low ceremony organizations). **Challenged**: previously approved or decided decision that is now in jeopardy; it may go back to approved without ceremony, but can also be demoted to tentative or rejected. **Rejected**: decision that does not hold in the current system; but we keep them around as part of the system rationale. **Obsolesced**: Similar to rejected, but the decision was not explicitly rejected (in favor of another one for example), but simply became ‘moot’, irrelevant as a result of some higher level restructuring, for example.

### 2.2.6 Categories

A design decision may belong to one or more categories. The list of categories is open ended; you could use them as some kind of keywords.

Categories will complement the taxonomy expressed above, if this taxonomy is not sufficient for large projects. (There is a danger in pushing taxonomy too far, too deep too early; it stifles creativity.) Categories are useful for queries, and for creating and exploring sets of design decisions that are associated to a specific concern or quality attribute. *Examples:*

– Usability
– Security

But the architects may be more creative and document also Politics: tagging decisions that have been made only on a political basis; it maybe useful to revisit them once the politics change. *Example:*

– “Use GIS Mapinfo” in Categories: politics, usability, safety, COTS

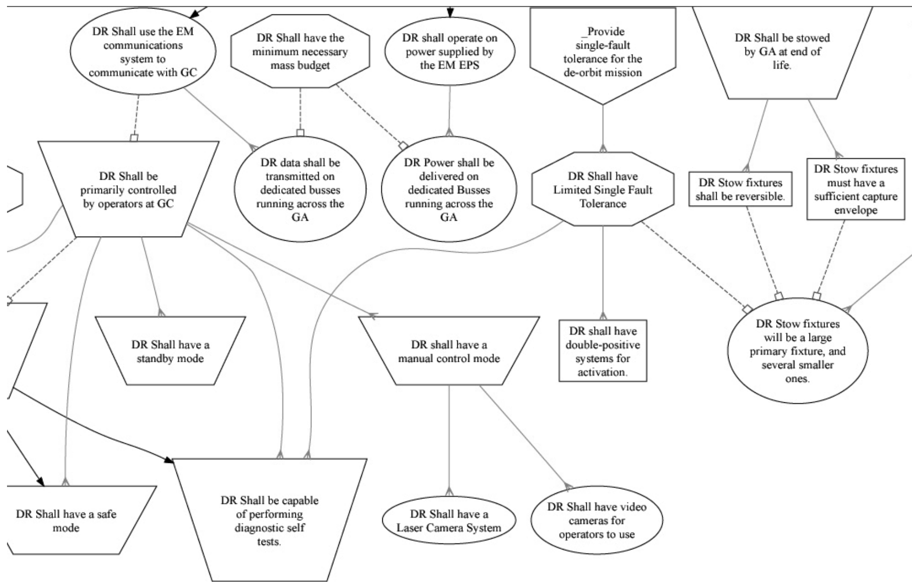


**Fig. 2.** Fragment of a decision graph for the SPAR Aerospace Dexterous Robotic Arm (DR). (Courtesy Nicolas Kruchten & Michael Trauttmansdorff)

### 2.3 Relationships Between Architectural Design Decisions

Decision A *"is Related to"* Decision B. This way, decisions form a graph-like structure. The use cases of section 3 refer to this structure, and the example in section 4 uses one. In the next subsections we discuss an initial set of relations between design decisions.

#### 2.3.1 Constrains

Decision B is tied to Decision A. If decision A is dropped, then decision B is dropped. Decision B is contingent to decision A, and cannot be promoted higher than decision A. The pattern is often that a property decision (rule or constraint) constrains an existence decision, or that an executive decision (process or technology) constrains a property decision or an existence decision. *Examples:*

– "Must use J2EE" constrains "use JBoss"; taking the dotNet route instead of J2EE would make JBoss the wrong choice.

#### 2.3.2 Forbids (Excludes)

A decision prevents another decision to be made. The target decision is therefore not possible. In other words, decision B can only be promoted to a state higher than 0 if decision A is demoted to a state of 0. (cf. section 2.2.5)

#### 2.3.3 Enables

Decision A makes possible Decision B, but does not make B taken. Also B can be decided even if A is not taken. It is a weak form of Constrains. *Example:*

– "use Java" enables "use J2EE"

### 2.3.4 Subsumes

A is a design decision that is wider, more encompassing than B. *Example:*

– "All subsystems are coded in Java" subsumes "Subsystem XYZ is coded in Java"

Often a tactical decision B has been made, which is later on generalized to A. It is often the case that the design decision could be reorganized to connect relatives of B to A, and to obsolesce B (B can be removed from the graph).

### 2.3.5 Conflicts with

A symmetrical relationship indicating that the two decisions A and B are mutually exclusive (though this can be sorted out by additional scoping decisions, cf. 0). *Example:*

– "Must use dotNet" conflicts with "Must use J2EE"

### 2.3.6 Overrides

A local decision A that indicates an exception to B, a special case or a scope where the original B does not apply. *Example:*

– "The Communication subsystem will be coded in C++" overrides "The whole system is developed in Java"

### 2.3.7 Comprises (Is Made of, Decomposes into)

A high level and somewhat complicated, wide-ranging decision A is made of or decomposes into a series of narrower, more specific design decisions B1, B2, … Bn. This is the case in high-level existence decisions, where partitioning or decomposing the system can be decomposed in one decision for each element of the decomposition. Or the choice of a middleware system, which implies a choice of various mechanisms for communication, error reporting, authentication, etc. This is stronger than constrains, in the sense that if the state of A is demoted, all the Bi are demoted too. But each B may be individually overridden.

Many of the rationale, alternatives etc. can be factored out and associated with the enclosing decision to avoid duplication, while details on a particular topic are documented where they belong. *Examples:*

– "Design will use UNAS as middleware" decomposes into

– "Rule: cannot use Ada tasking" and "Message passing must use UNAS messaging services" and "Error Logging must use UNAS error logging services" and etc.

### 2.3.8 Is an Alternative to

A and B are similar design decisions, addressing the same issue, but proposing different choices. This allows keeping around the discarded choices, or when brainstorming to relate the various possible choices.

Note that not all alternatives are conflicts, and not all conflicts are alternatives. But A conflicts with B is resolved by making A obsolete and an alternative to B.

### 2.3.9 Is Bound to (Strong)

This is a bidirectional relationship where A constrains B and B constrains A, which means that the fate of the two decisions is tied, and they should be in the same state.

### 2.3.10 Is Related to (Weak)

There is a relation of some sort between the two design decisions, but it is not of any kind listed above and is kept mostly for documentation and illustration reasons.

Examples are high level decisions that only provide the frame for other design decisions, while not being a true constraint (2.3.1) nor a decomposition (2.3.7).

#### 2.3.11 Dependencies

We say that a decision A depends on B if B constrains A (2.3.1), if B decomposes in A (2.3.7), if A overrides B (2.3.6). See figure 2 for an example of decision graph that depicts a number of such dependencies.

### 2.4 Relationship with External Artifacts

Decisions are not only related to other decisions, but also to other artifacts, such as requirements or parts of the implemented system (i.e. the Architectural Design, the models, the code). Example relationships in this category are `traces from' and `does not comply with'.

## 3 A Use Case Model for Architectural Knowledge

Assuming for a while that we have defined a repository of architectural knowledge in the form of all design decisions, how would we use it? Who would use it, to do what? Ultimately, every bit of architectural knowledge stored should be used in at least one use case and, conversely, every use case should be answerable from the architectural knowledge captured. A use-case model has at minimal *actors* (what are the various roles involved) and *use cases* (what do these roles do).

### 3.1 Actors

Who would use, produce, and exploit Architectural Knowledge from our repository?

- *Architects*: the people designing the system (or a part of a large system). They need to document much of the design, they should bring the decisions and assumptions from tacit to documented or formalized
- *Other architects*: People who are designing parts that integrate with that system. They need to understand the parts not directly under their responsibility, to see what impact it has on their decisions.
- *Developers*: People involved in the implementation of the design and decisions.
- *Reviewers*: people involved in judging the quality or progress of a design
- *Analysts*: Mostly, people dealing with requirements they are interested in
- *Maintainers*: while evolving or correcting the system they need to understand the correlation between decisions they take and the current set of decisions.
- *Users*: Not the end-users of the system, but people who use Architectural Knowledge, for example to interface another system, to document the system etc.
- *Re-Users*: people who want to exploit all or some of the Architectural Knowledge to build a new system
- *Students*: people who want to study software architecture by looking at Architectural Knowledge from various angles

- *Researchers*: Researchers may want to look at Architectural Knowledge to find new patterns, new information, better mousetraps.
- *Software tools*: Tools may both add to the AK repository, or exploit automatically some of the contents (consistency checking, pattern recognition, report generation, etc.)

From this list we can identify roles of *passive users or consumers* of Architectural Knowledge: people who need to exploit Architectural Knowledge for their own understanding but who are not going to alter it or expand it. Examples are Developers, Reviewers, and Students.

Other roles are those of *active users or producers* of architectural knowledge: they add to the Architectural Knowledge repository, integrate it, mature the information in it. Examples include Architects and Software tools.

### 3.2 Use Cases

From interviews held with practicing architects, as well as our own experience, we identified the following initial set of use cases:

- *Incremental architectural review:* what pieces of Architectural Knowledge have been added or modified since the last review? Extract and visualize these elements; browse and explore dependencies or traces.
- *Review for a specific concern:* from a given perspective (such as security, safety, reuse, etc.) what are the knowledge elements involved? This consists in building in some sense a “view” of Architectural Knowledge restricted to that concern.
- *Evaluate impact:* if we want to do a change in an element, what are the elements impacted (decisions, and elements of design). This may branch out to various kinds of changes: change of an assumption, change of a design decision.
- *Get a rationale*: given an element in the design, trace back to the decisions it is related to.
- *Study the chronology:* over a time line, find what the sequence of design decisions has been.
- *Add a decision:* manually or via some tool; then integrate the decision to other elements of Architectural Knowledge. (Similarly for other AK elements).
- *Cleanup the system:* make sure that all consequences of a removed decision have been removed.
- *Spot the subversive stakeholder:* identify who are the stakeholders whose changes of mind are doing the most damage to the system.
- Similar but different, *Spot the critical stakeholder:* the stakeholder who seems to have the most “weight” on the decisions, and who therefore maybe the one that could be most affected by the future evolution of the system.
- *Clone Architectural Knowledge:* produce a consistent subset of Architectural Knowledge to prime the pump for a new system (reuse Architectural Knowledge).
- *Integration:* you want to integrate multiple systems and decide whether they fit. The tool would help answering questions about integration strategies.
- *Detection and interpretation of patterns:* are there patterns in the graphs that can be interpreted in a useful fashion, and lead to guidelines for the architects. For example: decisions being hubs (“God” decisions), circularity, decisions that gain weight over time and are more difficult to change or remove.

## 4 Ontology-Based Information Visualization: A Case Study

Aduna Cluster Map Viewer[1] [8] is a tool to visualize ontologies that describe a domain through a set of classes and their hierarchical relationships. In such ontologies, classes often share instances: a software architect can be both a security expert and involved in project X. The Aduna Cluster Map Viewer is especially well suited to graphically depict such relationships. Based on a user query, it selects and displays the set of objects that satisfy the query. Objects belonging to the same cluster are depicted in one bubble. Clusters can belong to one or more classes.

Figure 3 shows part of the XML representation of the set of design decisions for the SPAR Aerospace Robotic Arm that is used as input to the Aduna Cluster Map Viewer.

```
- <ObjectSet>
  - <Object ID="obj2">
       <Name>Motion should be accurate to
            within +1 degree and +1 inch</Name>
    </Object>
  - <Object ID="obj33">
      <Name>Shall withstand all loads due to launch</Name>
    </Object>
  <\ObjectSet>
- <ClassificationSet>
  - <Classification ID="onto">
      <Name>Ontocrises</Name>
      <Objects objectIDs="obj2 obj4 … obj33 …" />
      <SuperClass refs="kind" />
  - <Classification ID="system">
      <Name>System</Name>
      <Objects objectIDs="… obj32 obj33 …" />
      <SuperClass refs="scope" />
    </Classification>
  - </Classification>
      <Name>Approved</Name>
      <Objects objectIDs="obj1 obj2 …" />
      <SuperClass refs="state" />
    </Classification>
<\ClassificationSet
```

**Fig. 3.** XML representation of design decisions. We have employed a taxonomy where the class DesignDecision has four subclassses: Kind (ontocrises, anticrises, etc; see section 2.1), Scope (System, Time, Organization; see section 2.2.3) and State (Idea, Tentative, etc; see section 2.2.5). We list here two design decisions, labeled obj1 and obj2. It further states that decisions labeled obj2, obj3 … are of type Ontocrises, while obj1, obj2 … have state Approved, Finally, decisions obj32, obj33 … have a System scope.

The same set of design decisions represented in XML in figure 3 is used to get the visualization of the example cluster map in Figure 4.

[1] http://aduna.biz/products/technology/clustermap

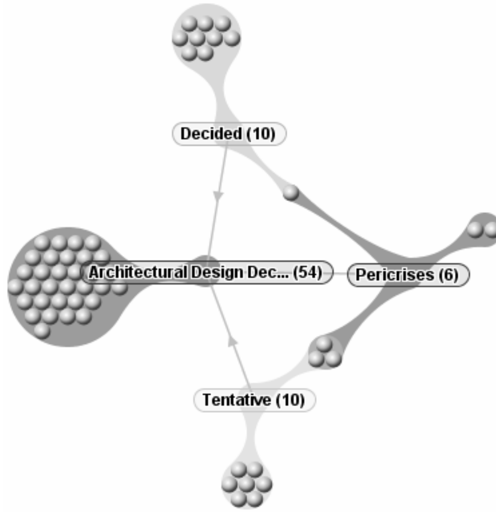


**Fig. 4.** The decisions are split into two types: pericrises and 'the rest'. For each of the decisions a state has been determined. State 'decided' contains the decisions that have been marked as decided. The figure shows the overlap between the clusters of all decisions, those that are pericrises, those that have been decided upon and those that are tentative. It shows that 10 decisions have been marked as decided, one of which falls in the class of pericrises. Of the remaining five pericrises, three are tentative.

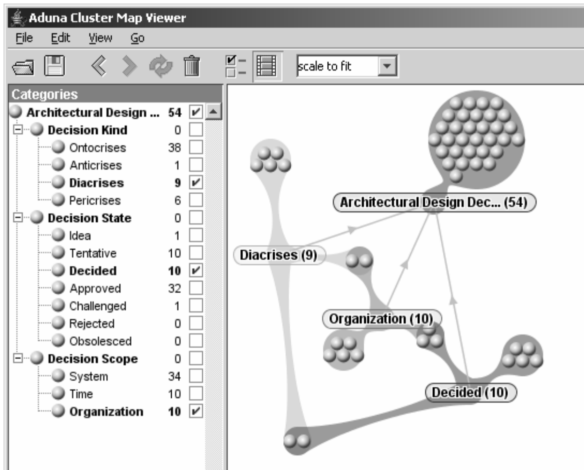


**Fig. 5.** Screenshot of Aduna Cluster Map Viewer. The left pane shows the various categories distinguished. It is generated from the XML file fed into the system. The user clicks the categories he is interested in. The right pane shows the results, including the intersections of the various sets. In this case, it shows the overlap between the clusters of all decisions, all diacrises, that are decided, and all organization-scoped decisions. A total of 10 decisions have been marked as decided, of which two fall in the class of diacrises, and three others are limited to organization scope. Of the remaining seven organization-scoped decisions, two are diacrises as well, but none of these are marked decided.

Figure 5 shows a screenshot of the Aduna Cluster Map Viewer. The selection panel on the left hand side is used to build a query whose result is depicted on the right hand side. Suppose we want to carry out some of the use cases in section 3.2. How would a tool like the Aduna Cluster Map Viewer aid the architecting process? For example, we can carry out 'clone AK' to create a new system by reusing all 'system' level decisions that are in state 'decided' (see Fig. 3). Use cases 'spot the critical stakeholder' and 'review for a specific concern' can be supported in a rather straightforward way by selecting all decisions that match a certain attribute value. 'Add a decision' or 'get a rationale' are realized by making selections or adding elements in the left pane (see Fig. 5).

This small case study shows that the Aduna Cluster Map Viewer can support many of our use cases in a rather straightforward way. A further discussion of tool support is given in section 5.

# 5 Conclusions

In this paper we discuss the notion of architectural knowledge, in particular the design decision part thereof. If we had a repository of architectural knowledge for a system, what would it ideally contain, how would we build it, and exploit it in practice? We describe a use case model for an architectural knowledge system, together with its underlying ontology. We present a small case study in which we model available architectural knowledge in a tool, the Aduna Cluster Map Viewer. Part of the use cases are handled by the tool, and part need further extension of the tool.

One of the most interesting unsolved issues is how to visualize architectural knowledge. The amount of information is overwhelming, so we have to abstract away from a lot of details. Developments in the area of information visualization are relevant here [7, 10, 27]. Most of these visualizations abstract away from details in the graph representation, and present the result in terms of tree maps, radial or conical representations, and the like. Fig. 2 gives an idea of a visualization of design decisions and constraints/requirements. In our opinion, though, this is not enough. The visualizations have to direct our attention to the very issue we want to convey, such as the subversive or critical stakeholder. For example, the Challenger accident in 1986 was ultimately due to low temperature. This problem had occurred before, but the link between low temperature and the rubber ring damage was not assessed and recognized until the right visualization was given [24].

If we can link our ontology of design decisions to the contents of actual design documents, we may exploit ontology-based browsers such as the Aduna Cluster Map Viewer [8] to explore these resources. This is a kind of data mining operation on documents containing architectural knowledge, with a targeted visualization. One such approach, using information retrieval techniques to obtain traceability information, is described in [11]. Assuming we have captured architectural knowledge as discussed above in some graph-like form, where the edges represent design decisions, and the vertices represent relationships between decisions, the different use cases correspond to certain operations on this graph. Some of these operations are relatively straightforward. They correspond to a subset operation (Clone AK, Review for a specific concern) or closure operation (Evaluate impact,

Cleanup the system). An interesting variation of change impact analysis using Bayesian Belief Networks is discussed in [23]. A more interesting operation has to do with the detection or matching of patterns or, rather, antipatterns. Some of these can be cast as graph operations; for example the detection of a “God” decision boils down to selecting nodes with a high fan-in/fan-out. The identification and operationalization of time-related patterns still is an open issue.

We might think of a series of plug-ins/services that each supports a specific use case and an appropriate visualization for that use case. For instance, the use case examples used in section 4 are all based on the analysis of flat subsets, and show that a tool like the Aduna Cluster Map Viewer might prove useful for browsing the design decision space. Instead, we might need another tool (and another type of visualization) to carry out use cases needing to traverse a hierarchical structure of design decisions and other knowledge entities. For instance, a tool visualizing directed graphs in two- or three-dimensional space seems better suited for use cases like 'evaluate impact' or 'study the chronology', which require to visualize not only categories of entities but also how these entities are related.

From this exercise, as well as our earlier experiences, we corroborate that tool support for manipulating architectural knowledge should focus on two crucial issues: suitable visualization, and task-specific support.

The list of use cases and associated operations needs further underpinning. We are currently in the process of validating and prioritizing this list with architects and development groups in various commercial settings.

## Acknowledgements

This research has partially been sponsored by the Dutch Joint Academic and Commercial Quality Research & Development (Jacquard) program on Software Engineering Research via contract 638.001.406 GRIFFIN: a GRId For inFormatIoN about architectural knowledge, and by an Eclipse Innovation grant from IBM.

## References

1. Bass, L., et al.: Software Architecture in Practice. Addison-Wesley, Reading, MA (2003).
2. Bosch, J.: Software Architecture: the Next Step. In *First European Workshop on Software Architecture (EWSA 2004)*, (2004), Springer-Verlag, 194-199.
3. Buckingham Shum, S. Analyzing the usability of a Design Rational Notation. In Moran, T.P. and Carroll, J.M. eds. *Design Rationale Concepts, Techniques, and Use*, Lawrence Erlbaum Associates, Mahwah, NJ (1996) 185-215.
4. Burge, J.E. and Brown, D.C. Reasoning with design rationale. In Gero, J.S. ed. *Artificial Intelligence in Design '00*, Kluwer Academic Publishers, Netherlands (2000) 611-629.
5. Clements, P., Bachmann, F., Bass, L., et al.: Documenting Software Architectures: Views and Beyond. Addison-Wesley, Boston (2002).
6. Conklin, J. and Begeman, M.L.: gIBIS: A tool for all reasons. *Journal of the American Society for Information Science*, 40 (1989).
7. Fekete, J.-D.: The InfoVis Toolkit. In *IEEE Symposium on Information Visualization 2004 (INFOVIS'04)*, (2004), 167-174.

8. Fluit, C., Sabou, M. and van Harmelen, F. Ontology-based information visualisation. In Geroimenko, V. and Chen, C. eds. *Visualising the Semantic Web*, Springer-Verlag (2005).
9. Fowler, M.: Who Needs an Architect. *IEEE Software*, 20 (5) (2003) 11-13.
10. Granitzer, M., Kienreich, W., Sabol, V., et al.: Evaluating a System for Interactive Exploration of Large, Hierarchically Structured Document Repositories. In *IEEE Symposium on Information Visualization 2004 (INFOVIS'04)*, (2004), IEEE CS, 127-133.
11. Hayes, J.H., Dekhtyar, A. and Sundaram, S.K.: Improving After-the-Fact Tracing and Mapping: Supporting Software Quality Predictions. *IEEE Software*, 22 (2005) 30-37.
12. IEEE standard 1471:2000--Recommended practice for architectural description of software intensive systems. *IEEE*, Los Alamitos, CA (2000).
13. Klein, M. DRCS: An Integrated System for Capture of Designs and Their Rationale. In Gero, J.S. ed. *Artificial Intelligence in Design '92*, Kluwer AP (1993) 393-412.
14. Kruchten, P.: The 4+1 View Model of Architecture. *IEEE Software*, 12 (6) (1995) 45-50.
15. Kruchten, P.: An Ontology of Architectural Design Decisions. In *2nd Groningen Workshop on Software Variability Management*, (2004), Rijksuniversiteit Groningen.
16. Lago, P. and van Vliet, H.: Explicit Assumptions Enrich Architectural Models. In *proceeding of ICSE 2005*, (2005), ACM Press, 206-214.
17. Lee, J.: Design Rationale: Understanding the Issues. *IEEE Expert 12* (1997) 78-85.
18. Lee, J.: SIBYL: a tool for managing group design rationale. In *ACM conference on Computer-supported cooperative work (CSCW90)*, (1990), 79 - 92.
19. Myers, K.L., Zumel, N.B. and Garcia, P.: Acquiring Design Rationale Automatically. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 14 (2000).
20. Nonaka, I., and Takeuchi, H.: The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation, Oxford University Press (1995).
21. Rus, I. and Lindvall, M.: Knowledge Management in Software Engineering. *IEEE Software*, 19 (2002) 26-38.
22. Tang, A., Babar, M.A., Gorton, I., et al.: A Survey of Architecture Design Rationale. In *WICSA 5*, (2005), IEEE CS.
23. Tang, A., Nicholson, A., Jin, Y., et al.: Using Bayesian Belief Networks for Change Impact Analysis in Architecture Design. In WICSA 5, (2005), IEEE CS.
24. Tufte, E.R.: Visual explanations: images and quantities, evidence and narrative. *Graphics Press LLC*, Cheshire, CO (1997).
25. Tyree, J. and Akerman, A.: Architecture Decisions: Demystifying Architecture. *IEEE Software*, 22 (2005) 19-27.
26. van der Ven, J.S., et al. Using Architectural decisions. In Hofmeister, C., Crnkovic, I., Reussner, R. and Becker, S. eds. Perspectives in Software Architecture Quality, Universitaet Karlsruhe, Fakultaet fuer Informatik (2006).
27. van Ham, F.: Using Multilevel Call Matrices in Large Software Projects. In *IEEE Symposium on Information Visualization 2003 (INFOVIS'03)*, (2003), IEEE CS, 227-232.