Chapter 3
Documentation of Software Architecture
from a Knowledge Management Perspective – Design Representation

Philippe Kruchten

Abstract In this chapter we survey how architects have represented architectural knowledge and in particular architectural design. This has evolved over the last 3 decades, from very intuitive and informal, to very structured, abstract and formal, from simple diagrams and metaphors, design notations, and specific languages. As our understanding of architectural knowledge evolved, the importance of design rationale and the decision process became more and more prominent. There is however a constant through this evolution: the systematic use of metaphors.

3.1 Introduction

When we speak about the documentation of software architecture, we are clearly referring to the explicit aspect of architectural knowledge (see Sect. 1.2.2.1). “If it is not written, it does not exist,” I used to tell members of an architecture team, prodding them to document, i.e., make very explicit, what we had discussed at length in a meeting, or the knowledge they had brought in from outside, their past experience in general, and especially the decisions we had just made.

Architectural knowledge consists of architectural design – the blueprints of the system under development – as well as the design decisions, assumptions, context, and other factors that together determine why a particular final solution is the way it is. Except for the architecture design part, most of the architectural knowledge usually remains hidden and tacit – in the heads of the architects. An explicit representation of architectural knowledge is helpful for building and evolving quality systems [192].

In this chapter, we will therefore focus on the lower right of the four quadrants of knowledge described in Fig. 2.1, the externalized part of architectural knowledge (as opposed to tacit knowledge): the application-specific explicit knowledge.

Philippe Kruchten (✉)
University of British Columbia, Vancouver, Canada, e-mail: pbk@ece.ubc.ca

M. Ali Babar et al. (eds.), Software Architecture Knowledge Management, DOI: 10.1007/978-3-642-02374-3_3, © Springer-Verlag Berlin Heidelberg 2009
and, to a somewhat lesser extent, we will also focus on the lower left quadrant: the application-generic explicit knowledge.

Architecture representation implies the use of models, architectural models. But what is a model? M is a model of S if M can be used to answer questions about S, where S is the system under consideration.

3.2 Evolution of Architectural Representation

The first reference to the phrase “software architecture” occurred in 1969 at a conference on software engineering techniques organized by NATO [65]. Some of our field’s most prestigious pioneers, including Tony Hoare, Edsger Dijkstra, Alan Perlis, Per Brinch Hansen, Friedrich Bauer, and Niklaus Wirth, attended this meeting. From then until the late 1980s, the word “architecture” was used mostly in the sense of system architecture (meaning a computer system’s physical structure) or sometimes in the narrower sense of a given family of computers’ instruction set. Key sources about a software system’s organization came from Fred Brooks in 1975 [54], Butler Lampson in 1983 [198], David Parnas from 1972 to 1986 [244, 245, 246, 247], and John Mills whose 1985 article looked more into the process and pragmatics of architecting [227]. The concept of software architecture as a distinct discipline started to emerge in 1990 [297]. A 1991 article by Winston W. Royce and Walker Royce (father and son) was the first to position software architecture – in both title and perspective – between technology and process [269]. Eberhardt Rechtin dedicated a few sections to software in his 1991 book *Systems Architecting: Creating and Building Complex Systems* [263].

3.2.1 Boxes and Arrows

In the 1970s and through most of 1980s, because the concept of software architecture was not very well delineated from that of software design or “high-level design”, there was very little agreement on how to document software architecture. Various mixes of “boxes and arrows” diagrams were used as models, far too often with not much precise semantics behind the boxes, and even less behind the arrows. Some of this remains today, and constitutes what I call the “PowerPoint” level of software architecture documentation. It is still very valuable in bridging the gaps between various groups of stakeholders (the various people the architect has to deal with), such as marketing, sponsors, quality, process, approval, certification, etc.

3.2.2 Views

Still today, modern software architecture practices rely on the principles that Perry and Wolf enounced in the late 1980s in their pretty and simple formula: Architecture = {Elements, Form, Rationale} [250]. The elements are the main constituents
of any architectural description in terms of components and connectors, while the nonfunctional properties guide the final shape or form of the architecture. Different shapes with the same or similar functionality are possible, as they constitute valid design choices by which software architects make their design decisions. These decisions are precisely the soul of architectures, but they are often neglected during the architecting activity as they usually reside in the architect's mind in the form of tacit knowledge that is seldom captured and documented in a usable form. Software architecture started to take shape as an artifact of the design process that "encompasses significant decisions about:

1. The organization of a software system
2. The selection of the structural elements and their interfaces by which a system is composed with its behavior as specified by the collaboration among those elements
3. The composition of these elements into progressively larger subsystems" (from RUP [189, 152])

For years, the generalized practice and research efforts have focused solely on the architectural representation itself. These practices have long been exclusively aimed at representing and documenting the system's architecture from different perspectives called architectural views. These views, which represent the interests of different stakeholders, are offered as a set of harmonized descriptions in a coherent and logical manner and also used to communicate the architecture.

3.2.3 The Architecting Process

The period between 1996 and 2006 brought complementary techniques in the form of architectural methods, many of them derived from well established industry practices. Methods like RUP (at Rational, then IBM) [189, 152], BAPO/CAFR (at Philips) [237], S4V (at Siemens) [147, 300], ASC (at Nokia), ATAM, SAAM and ADD (at the SEI) [175], among others, are now mature design and evaluation practices to analyze, synthesize, and evaluate modern software architectures. In some cases, the methods are backed by architectural description languages, assessment methods, and stakeholder-focused, decision-making procedures.

Since many of the design methods were developed independently [146], they exhibit certain similarities and differences motivated by the different nature, purpose, application domain, or the size of the organization for which they were developed.

In essence, they cover essential phases of the architecting activity but are performed in different ways. Common to some of these methods is the use of design decisions that are evaluated during the construction of the architecture. These decisions are elicited by groups of stakeholders, under the guide of architects, but the ultimate decision makers are (a small group – often a single person) architects. Unfortunately, design decisions and their rationale were still not considered as first-class entities because they lack an explicit representation. As a result, software architects cannot revisit or communicate the decisions made, which in most cases vaporize forever.
3.2.4 Architectural Design Decisions

Rationale was present in the 1992 formula of Perry and Wolf, but in reality, it was rarely captured explicitly in a form that would allow it to be revisited. This remains a prevalent issue.

Rus and Lindvall wrote in 2002 that “the major problem with intellectual capital is that it has legs and walks home every day” [272]. Current software organizations suffer the loss of this intellectual capital when experts leave. The same happens in software architecture when the reasoning required for understanding a particular system is unavailable and has not been explicitly documented. In 2004, Jan Bosch stated that “we do not view a software architecture as a set of components and connectors, but rather as the composition of a set of architectural design decisions” [48, 161]. The lack of first-class representation of design rationale in current architecture view models brought the need to include decisions as first-class citizens that should be embodied within the traditional architecture documentation.

There are several benefits of using design rationales in architecture as a mean to explain why a particular design choice is made or to know which design alternatives have been evaluated before the right or the optimal design choices are made. Benefits can be achieved in the medium and long term because documenting design rationale prevents the need for architecture recovery processes, which are mostly used to retrieve the decisions when design, documentation, or even the creators of the architecture are no longer available. In other cases, the natural evolution of a software system forces previous design decisions to be replaced by new ones. Hence, maintaining and managing this architectural knowledge requires a continuous attention to keep the changes in the code and the design aligned with the decisions, and to use these to bridge the software architecture gap.

3.2.5 Architectural Knowledge = Architectural Design + Architectural Design Decisions

It is in this new context that Perry and Wolf’s old ideas [250] become relevant for upgrading the concept of software architecture by explicitly adding the design decisions that motivate the creation of software designs. Together with design patterns, reference architecture, frameworks, etc., design decisions are a subset of the overall architectural knowledge (AK) that is produced during the development of architecture. Most of the tacit knowledge that remains hidden in the mind of the architects should be made explicit and transferable into a useful form for later use, easing the execution of distributed and collective decision-making processes.

The formula, Architectural Knowledge = Architectural Design + Architectural Design Decisions, recently proposed by Kruchten, Lago and van Vliet [192], modernizes Perry and Wolf’s [250] idea and considers design decisions as part of the architecture. We’ll use this formula: AK = AD + ADD, to structure the rest of this chapter.
3.3 Architectural Design

For many years, architectural knowledge was represented in very ad hoc fashion: boxes and arrows of unspecified or fuzzy semantic value; the PowerPoint level of architectural description. The challenge was to describe in a truly multidimensional reality in a 2-dimensional space. Because the model, M, was poor, the set of questions that could be answered regarding the system it represented remained limited.

3.3.1 Viewpoints and Views

In the early 1990s, several groups around the world realized that the architecture of a large and complex software-intensive system was made of multiple entangled structures, and that the poor attempts of representing architecture using a single type of flat blueprint was inherently doomed to fail. Different parties (or stakeholders) are concerned by different aspects of an architecture. Each of these aspects is a viewpoint, and can be addressed by a certain kind of architecture representation, with a notation and language of its own. The views, however, do not correspond to a decomposition of the architecture in parts, but are different projections, abstractions or simplifications of a more complex reality.

Figure 3.1 is an example of a set of five views (from [152] and [188]). Very similar sets of views appeared at Siemens [300] and elsewhere.

This concept of multiple views is not new, and Paul Clements traced it to a 1994 paper by David Parnas: “On a buzzword: hierarchical structure” [245]. An architectural view is a partial representation of a system, from the perspective of a

![Diagram](image-url)
well-defined set of architectural concerns. A viewpoint is a set of conventions for the construction, interpretation, and use of a given architectural view. In a way, the viewpoint is to a view what the legend is to a map [158]. Of great importance are the view correspondences, that is, the relationships that exist between the elements in one view and the elements in another view.

Much has been written on the concept of views for architectural description, in particular by a group at the SEI led by Paul Clements [71] in their 2003 book *Documenting Software Architecture: Views and Beyond* and by Rozanski and Woods [270] who introduced the refinement of perspective. The IEEE standard 1471–2000 [155] provides a guide for describing the architecture of complex, software-intensive systems in terms of views and viewpoints, but it does not offer a detailed description of the rationale that guides the architecting process (see below).

### 3.3.2 Architecture Description Languages

Over the last 20 years computer scientists in academia around the world have tried to create architecture description languages (ADLs) to capture, represent, and reason about essential elements of the conceptual architecture of a software-intensive system. This follows the long tradition of programming languages: no computer scientist would be “complete” who has not created his or her own language, and the compiler that goes with it.

What ADLs have in common is that they are able to denote essential elements of an architecture – components, packages, and to a certain extent some of the characteristics and behavior of these components and connectors – so that some form of analysis, verification, and reasoning can occur, to derive or assess completeness, consistency, ambiguity, performance, and other qualities. ADLs often offer both a textual and a graphical notation, the intention being to have them readable by both humans and machines.

ADLs have evolved from Rapide at Stanford, to ACME (CMU), Wright (CMU), C2 (UCI), Darwin (Imperial College), to name only a few. Koala – developed by Philips and based on Darwin – is the only one that has had some industry penetration, and is used primarily to configure product-line instances in the consumer electronics domain. Also AADL was developed by the SAE based on MetaH for the automotive industry.

Finally, there has been some heated debate on the question of whether or not the unified modeling language (UML) is an ADL. While not limited to architecture, UML 2.0 certainly has all the elements to describe architecture. It is the notation used to represent the 4+1 views above [152].

As an ADL is a notation to express an architectural model, encompassing the key design decisions that bind the system, it should be an important tool for the capture and representation of architectural knowledge. Unfortunately, so far, except for UML, ADLs’ use has been very limited and mostly confined to academic labs.
3.3.3 Application-Generic Knowledge: Patterns, Standards, Frameworks

When moving from application-specific to application-generic knowledge, across a domain or a technology or simply a community of architects, some application-generic architectural knowledge can be developed and captured, using pretty much the same tools and techniques we’ve seen above in Sect. 3.3.1 and 3.3.2.

Patterns and Frameworks

Tagging behind the work of the famous “Gang of Four”, Buschmann and his gang of five captured architectural patterns [64] followed by a few others: SEI [74]. These patterns are technology neutral. But others can be technology and/or domain specific, such as the architectural patterns developed in Microsoft’s practice and patterns group’s handbook [225]. More ambitious, an architectural framework is a set of common practices for architectural description established within a specific domain or stakeholder community; it identifies generic concerns, and some predefined viewpoints which frame these concerns. Examples of frameworks are TOGAF (The Open Group Architectural Framework), MoDAF, the Zachman Framework, RM ODP and ISO 19439 (FDIS).

Standards

IEEE Std 1471 [155] was the first formal standard for architectural description, in active use since 2000. In 2007, IEEE 1471 became an international standard. Now ISO and IEEE are jointly revising the standard as ISO/IEC 42010, Systems and Software Engineering – Architecture Description [158].

There are new knowledge mechanisms in ISO/IEC 42010. In a sense, every standard is knowledge-based, embodying a community consensus by creating a filter on the world through its definitions, and establishing rules on what to do when those definitions apply. An important element of IEEE 1471:2000 was the explicit conceptual model (or ontology) upon which the standard was built. That model has been useful in codifying architectural practice, in education and in advancing the state of the practice. The most obvious knowledge mechanisms in ISO/IEC 42010 are those reflecting resources readily reusable by architects from one project to another: architecture-related concerns, stakeholder identification, and architectural viewpoints.

In the ongoing revision (working draft 3, Nov 8, 2008), ISO/IEC 42010 [158] is considering several new mechanisms:

• Codifying architectural frameworks: for large-scale reuse and knowledge sharing
• View correspondences: for linking between views
• Architectural models and model types: for finer-grain reuse
• Enhanced architecture rationale and decision capture

where IEEE 1471 codified a terminology base and best practices applicable to individual architecture descriptions, ISO/IEC 42010 introduces a further level of conformance in defining the notion of an architecture framework, which we define here as a set of best practices for a particular community or domain, characterized...
by a set of concerns, stakeholders, viewpoints, and the correspondences between those viewpoints.

From a knowledge perspective, the hope is that many of the practices currently called architecture frameworks in the community can now be defined in a uniform way, thereby raising the level of understandability and interoperability – some might say “reusability” – among architects working within different paradigms. One mechanism added to support architecture frameworks is view correspondence rules, a notion that was not ready for standardization in 2000 (R. Hilliard, personal communication).

**Methods**

Finally, methods pertaining to software architecture do also capture some architectural knowledge. This is the case for the Siemens method [147], IBM’s RUP [152], the many SEI methods: ADD, ATAM, QAW [177, 34] or the *Software Architecture Review and Assessment* handbook [236], for example.

### 3.4 Architectural Design Decisions

#### 3.4.1 What Is an Architectural Design Decision?

In his 2003 paper, Jan Bosch [48] stressed the importance of design decisions as “a first class citizen”, but did not describe in detail what they consist of. Tyree and Ackerman describe the structure of an architectural design decision (see Table 1.1 in Chap. 1) [325]. Kruchten in [190], then with Lago and van Vliet in [192], introduce a more detailed template for documenting design decisions, particularly stressing relationships between design decisions and between design decisions and other artifacts. Tang [314], Dueñas and Capilla [66, 103] have variations, and the SHARK workshop in 2006 attempted to reconcile all these views.

Here is an example of attributes of an architectural design decision, from [190] and [192]:

- **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, etc.
- **Rationale.** This is a textual explanation of the “why” of the decision, its justification. Care should be taken not to simply paraphrase or repeat information captured in other attributes, but to add value. If the rationale is expressed in a complete external document, for example, a tradeoff analysis, then the rationale points to this document.
- **Scope.** Some decisions may have limited scope, in time, in the organizations, or in the design and implementation (see the overrides relationship below). By default, (if not documented) the decision is universal. Scope might delimit the part of the system, a life cycle time frame, or a part of the organization to which the decision applies.
• **State.** Like problem reports or code, design decisions evolve in a manner that may be described by a state machine (see Fig. 3.2):

  – **Idea.** Just an idea, captured so it is not lost, when 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.** Approved by a review, or a board (for low-ceremony organizations, not significantly different than decided).
  – **Challenged.** A 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 status.
  – **Rejected.** A decision that does not hold in the current system; but we keep them around as part of the system rationale (see subsumes below).
  – **Obsolesced.** Similar to rejected, but the decision was not explicitly rejected (in favour of another one, for example), but simply became “moot,” irrelevant, e.g., as a result of some higher-level restructuring.

• **Author, Time-stamp, History.** The person who made the decision and when. Ideally we collect the history of changes to a design decision. State changes are important, but so are changes in formulation and scope, especially with incremental architectural reviews.

• **Categories.** A design decision may belong to one or more categories. The list of categories is open ended. Categories are useful for queries and for creating and exploring sets of design decisions that are associated with specific concerns or quality attributes.

• **Cost.** Some design decisions have an associated cost, so it is useful to reason about alternatives.
• **Risk.** Traditionally documented by exposure – a combination of *impact* and *likelihood* factors – this is the risk associated with taking a particular decision (see IEEE Std 1540-2001, for example). It is often related to the uncertainty in the problem domain, or to the novelty of the solution domain, or to unknowns in the process and organization. If the project is using a risk management tool, this should simply link to the appropriate risk in that tool.

• **Related decisions.** Decision A “is related to” decision B in any of the following ways:
  
  – *Constrains:* “must use J2EE” constrains “use JBoss”.
  – *Forbids* (or excludes): “sin single point of failure” forbids “use a central server”.
  – *Enables:* “use Java” enables “use J2EE”.
  – *Subsumes:* “all subsystems are coded in Java” subsumes “subsystem XYZ is coded in Java”.
  – *Conflicts with:* “must use dotNet” conflicts with “must use J2EE”.
  – *Overrides:* “the Comm subsystem will be coded in C++” overrides “the whole system is developed in Java”.
  – *Comprises* (is made of, decomposes into): “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.
  – *Is Bound to* (strong): A constrains B and B constrains A.
  – *Is an Alternative to:* A and B address the same issue, but propose different choices.
  – *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.

See Fig. 3.3 for an example.

• **Relationship with External Artifacts.** Includes “traces from,” “traces to,” and “does not comply with.” Design decisions trace to technical artifacts upstream:

![Fig. 3.3 Example of relationship between design decisions, from [203]](image-url)
requirements and defects, and artifacts downstream: design and implementation elements. They also trace to management artifacts, such as risks and plans. These relationships are almost as important as the decisions themselves. It is through these relationships that decisions can be put to practical use: understanding part of the rationale – the reason for certain decisions, for the choices made from among several alternatives, for the incompatibilities between choices – impacts the analysis of “what is affected if we were to change X, or Z?”

3.4.2 A Taxonomy of Architectural Design Decisions

Architectural design decisions do not all play the same role in the architecting process. Some are tightly coupled to the design itself, and can be traced directly to some element (e.g., a class, a process, a package or subsystem, an interface) in the system under development; other decisions are general properties or constraints that we impose to the system, that sets of elements must satisfy, and, finally, some are linked to the general political, sociological, and cultural environments of the development or deployment.

3.4.2.1 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, and 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 nonfunctional requirement (quality attribute) or connector. Examples:

- Dextrous Robot (DR) shall have a Laser Camera System.
- DR shall use the Electromagnetic (EM) communication system to communicate with Ground-Control.

In themselves, existence decisions are not 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 to alternatives (see Fig. 2.1).
3.4.2.2 Bans or Nonexistence Decisions ("Anticrises")

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

It is important to document bans precisely because such decisions are lacking any "hooks" in traditional architecture documentation. They are not traceable to any existing artifact. Ban decisions are often made as possible alternatives are gradually eliminated.

3.4.2.3 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), of a trait that the system will not exhibit. Properties are harder to trace to specific elements of the design or 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.

3.4.2.4 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 [155]. There are all the political, personal, cultural,
There are two delicate issues with sets of architectural decisions:

- How to capture them (and how much)?
- How to visualize them?

Capture is a process or method issue and will be covered in a subsequent chapter. How many decisions and how much information must be captured are hard questions that relate to the other, more fundamental, question: what is the scope of architecture? In Chap. 1 we stressed that architecture is a global, high-level, early process, aimed at making hard, long-lived, and hard-to-change choices. Still pertinent to the issue of knowledge representation is: how do we represent sets of interrelated decisions?

Assuming that we have captured a set of architectural design decisions, along the lines of Sect. 3.4.1 above, how would we want to visualize them? The tabular approach is not very exciting (see Fig. 3.4).

We can also represent architectural design decisions as graphs, stressing the relationships we have described in Sect. 3.4.1, but these graphs rapidly become very complex (see Fig. 3.5), and we need to introduce mechanisms for:

- **Eliding** (eliminating certain relationships), or zooming in and out (getting more or less information about decisions) (see Fig. 3.6).

Fig. 3.4 The tabular representation (from [202])
Filtering (limiting the number of decisions in a view based on a set of criteria) or clustering (grouping decisions according to some criteria) (see Fig. 3.7).

Focusing (using some decision as an anchor, or a center for displaying other decisions).

Sequencing (laying out on a time line).

In particular, the focus on one particular decision supports the concept of impact analysis: show me all the decisions that are affected by this one decision (see Fig. 3.8) [190, 193, 201, 203, 204].

A sequence of design decisions over a period of time supports the concept of incremental architectural review: what are the decisions that have been made or changed since the last review?
3.4.4 A “Decisions View” of Architecture

If views and viewpoints are a good practice to document the design, and if a set of design decisions offers a good complement in capturing (externalizing) additional architectural knowledge, then there might be a way to combine the two approaches for more cohesiveness and homogeneity. This is what Dueñas and Capilla [103] have done in proposing a decision view of software architecture in which decisions are entangled with design for each architectural view.

This new perspective extends the traditional views that are described in the IEEE Std. 1471 [155] by superimposing the design rationale that underlies and motivates the selection of concrete design options. Figure 3.9 depicts a graphical sketch of the “decision view” [103, 191] in which design decisions are attached in the “4 + 1” view model [188].

Fig. 3.7 Clustering decisions by classes (from [192])
Fig. 3.8 Impact analysis, starting from decision #11 [203]

Fig. 3.9 A decision view embedded in the 4 + 1 views (from [191])
3.5 Rationale, or, the Missing Glue

Rationale, that is, an explicit articulation of the reasoning, the motivation of the choice implied by the decision, has been present in the mind of software architect all along; it was explicit in the formula of Perry and Wolf in 1992 [250]. The rationale can range from a simple pointer to a requirement to an elaborate trade-off analysis between alternative solutions. Often, it also has to show that various constraints or previous decisions are taken into account. But, in practice the effort to develop tools to capture and manage design rationale has not been very successful, as Jintae Lee eloquently described [200]. The primary reason is that capturing rationale, except for a handful of important decisions; it is too tedious and does not bring any immediate benefits to the person doing the capture. The benefit is largely down the line, weeks or months later and for stakeholders others than the decision maker.

We found that much of the rationale is actually captured by the relationship between design decisions (DDs) and other elements, in particular by tracing DDs to requirements (upstream), to other decisions (see above), and to elements in the design and its implementation (downstream).

3.6 Metaphors

There is one constant, though, throughout the whole (short) history of software architecture, and regardless of the formality of the approach: it is the systematic use of metaphors to describe architectural elements an architectures. Metaphors give meaning to form and help us ground our conceptual systems. A metaphor is a form of language that compares seemingly unrelated subjects: a rhetorical trope that describes a first subject as being equal or very similar to a second object in some way. A metaphor implies a source domain: the domain from which we draw metaphorical expressions, and a target domain, which is the conceptual domain we try to understand or to describe. The metaphor operates a cognitive transfer from the source to the target; it says in essence: "<target> is <source>.

In *Metaphors we live by* [197], Lakoff and Johnson describe metaphors as "a matter of imaginative rationality." They permit "an understanding of one kind of experience in terms of another, creating coherence by virtue of imposing gestalts that are structured by natural dimensions of experience. New metaphors are capable of creating new understanding and, therefore, new realities." (p. 235)

Metaphors are everywhere in software architecture. We use so-called *ontological metaphors* to name things: "clients and servers", "layers", "pipes and filters", "department stores and shopping cart," etc. We organize those using *structural metaphors* that are often visual and spatial: "on top of", "parallel to", "aligned with", "foreground, background", but include richer ones such as "network", "web", or "hierarchy" [245]. We use a wide variety of containers: "packages", "repositories", "libraries", "volumes", etc. In reality, in the target domain, i.e., in the memory of
a computer, we would not find any up, down, aligned, packaged, etc. everything is
pretty scattered around; just look at a “core dump” file.

A mapping is the systematic set of correspondence that exists between con-
stituent elements of the source and the target domains. It allows some limited
reasoning in the target domain by analogy and inference. In our case, the tar-
get domain – software architecture – is rather abstract, and we try to draw from
source domains that are much more concrete. Then we use inference patterns from
the source conceptual domain to reason about the target one. “Ay, there’s the rub,
for” we may abuse the inference or have a faulty inference. This leads to flawed
metaphors, where the analogy “breaks” and the meaning in the target domain (in
software) is confusing at best. It is also very common when we attempt to combine
multiple metaphors, drawing form different source domains.

Metaphors have been used to describe general organization of software systems;
they have played an important role in the object-oriented movement, and then were
really put at the center of the pattern movement. No reasonable pattern can be suc-
cessful that is not supported by an elegant metaphor [64, 130]. More recently Beck
in eXtreme Programming (XP) described a practice he called “metaphor” to con-
vey the concept of a simple summary of the architecture [38]. It is unfortunate that
this practice has been the least successful of XP. Beck should have boldly called it
Allegory: an extended metaphor in which a story is told to illustrate an important
attribute of the subject, or even a Parable!

3.7 Summary

To return to our premises: Why do we want to represent architectural knowledge
explicitly? We want to:

• Gain intellectual control over a sophisticated system's enormous complexity.
• Ensure the continuity, allowing these large systems to more effectively evolve and
to be maintained.
• Transfer this knowledge to others.

More tactically, we want to be able to

• Analyze and evaluate architectures, implement them, evolve them, assess some
  of their qualities
• Support planning, budgeting, or acquisition activities

The more this architectural knowledge is left implicit, the more difficult or risky
these activities will be. Moreover, we also want to be able to abstract some more
generic architectural knowledge out of our collective knowledge in a given domain
or with given technologies. This knowledge is a combination of the following.

1. Architectural design, generic, brought in by using expert architects, educa-
tion, framework, methods, and standards. They are templates, exemplars of our
models.
3. Documentation of Software Architecture

2. Architectural design, for the system under development, expressed using a combination of appropriate notations and tools, adapted to the concerns of the various parties involved (viewpoints). They are models of the system.

3. Architectural decisions, for the system under development, with their complex interdependencies, and tracing upstream to requirements, context, and constraints, and tracing downstream to the design, and even the implementation. They explain why the models are the way they are.