Architectural design decisions

Jansen, Antonius Gradus Johannes

CHAPTER 3

DESIGN DECISIONS: THE BRIDGE BETWEEN RATIONALE AND ARCHITECTURE

Published as: Jan S. van der Ven[1], Anton G. J. Jansen[1], Jos A. G. Nijhuis, Jan Bosch. Design Decisions: The Bridge between Rationale and Architecture, chapter 16 , Rationale Management in Software Engineering, Allen H. Dutoit, Raymond McCall, Ivan Mistrik, Barbara Paech Editors, pp. 329-346 , Springer-Verlag, April 2006.

Abstract

Software architecture can be seen as a decision making process; it involves making the right decisions at the right time. Typically, these design decisions are not explicitly represented in the artifacts describing the design. Instead, they reside in the minds of the designers and are therefore easily lost. Rationale management is often proposed as a solution, but lacks a close relationship with software architecture artifacts. Explicit modeling of design decisions in the software architecture bridges this gap, as it allows for a close integration of rationale management with software architecture. This improves the understandability of the software architecture. Consequently, the software architecture becomes easier to communicate, maintain and evolve. Furthermore, it allows for analysis, improvement, and reuse of design decisions in the design process.

[1] Both authors contributted equally to this paper

3.1 Introduction

Software design is currently seen as an iterative process. Often used phases in this process include: requirements discussions, requirements specification, software architecting, implementation and testing. The Rationale Unified Process (RUP) is an example of an iterative design process split into several phases. In such an iterative design process, the software architecture has a vital role [119].

Architects describe the bare bones of the system by making high-level design decisions. Errors made in the design of the architecture generally have a huge impact on the final result. As such, a lot of effort is spent on making the right design decisions in the initial design of the architecture. However, the rationale underlying the architecture is not usually documented, because the focus is only on the results of the decisions (the architectural artifacts). Therefore the evaluated alternatives, tradeoffs, and rationalisations about the made decisions remain in the heads of the designers. This tacit knowledge is easily lost. The lost architecture knowledge leads to evolution problems [168], increases the complexity of the design [21], and obstructs the reuse of design experience [93].

To solve the problem of lost architectural knowledge, techniques for managing rationale are frequently proposed. Experiments show that maintaining rationale in the architecture phase increases the understandability of the design [22]. However, creating and maintaining this rationale is very time-consuming. The connection to the architectural and design artifacts is usually very loose, making the rationale hard to use and keep up-to-date during the evolution of the system. Consequently, there seems to be a gap between rationale management and software architecture.

To bridge this gap, we unite rationale and architectural artifacts into the concept of a design decision, which couples rationale with software architecture. Design decisions are integrated with the software architecture design. By doing this, the rationale stays in the architecture, making it easier to understand, communicate, change, maintain, and evolve the design.

Section 3.2 of this chapter introduces software architectures. Section 3.3 discusses how rationale is used in software architectures. Section 3.4 introduces the concept of design decisions. Section 3.5 presents a concrete approach that uses this concept. After this, related and future work is discussed, followed by a summary, which concludes this chapter.

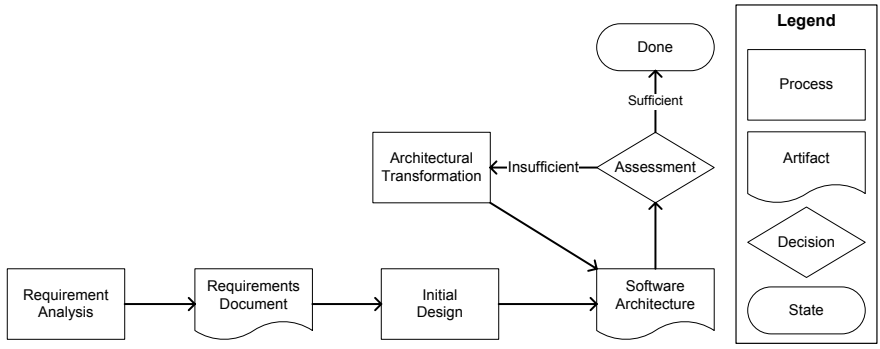

Figure 3.1: An abstract view on the software architecture design process

3.2 Software architecture

This section focuses on the knowledge aspects of software architectures. In subsection 3.2.1, the software architecture design process is discussed. Next, different ways are presented to describe software architectural knowledge in subsection 3.2.2. Subsequently, the issue of knowledge vaporization in software architecture is discussed in subsection 3.2.3.

3.2.1 The software architecture design process

A software architecture is based on the requirements for the system. Requirements define what the system should do, whereas the software architecture describes how this is achieved. Many software architecture design methods exist (e.g. [11] and [19]), and they all use different methodologies for designing software architectures. However, they can all be summarized in the same abstract software architecture design process.

Figure 3.1 provides a view of this abstract software design process[2] and its associated artifacts. The main input for a software architecture design process is the requirements document. During the initial design the software architecture is created, which satisfies (parts of) the requirements stated in the requirement document. After this initial design phase, the quality of the software architecture is assessed. When the quality of the architecture is not sufficient, it is modified (architectural modification).

[2]Remark this is a rather simplistic view, as the iterative interation with requirements is missing. Chapter 7 on page 143 provides a more detailed and precise view.

To modify the architecture, the architect can among other things, employ a number of tactics [11] or adopt one or more architectural styles or patterns [138] to improve the design. This is repeated until the quality of the architecture is assessed sufficient.

3.2.2 Describing Software Architectures

There is no general agreement of what a software architecture is and what it is not. This is mainly due to the fact that software architecture has many different aspects, which are either technically, process, organization, or business oriented [19]. Consequently, people perceive and express software architectures in many different ways.

Due to the many different notions of software architectures, a combination of different levels of knowledge is needed for its description. Roughly, the following three levels are usually discerned:

Tacit/Implicit knowledge In many cases, (parts of) software architectures are not explicitly described or modeled, but remain as tacit information inside the head(s) of the designer(s). Making this implicit knowledge explicit is expensive, and some knowledge is not supposed to be written down, for example for political reasons. Consequently, (parts of) software architectures of many systems remain implicit.

Documented knowledge Documentation approaches provide guidelines on which aspects of the architecture should be documented and how this can be achieved. Typically, these approaches define multiple views on an architecture for different stakeholders [70]. Examples include: the Siemens four view [67], and the work of the Software Engineering Institute [29].

Formalized knowledge Formalized knowledge is a specialized form of documented knowledge. Architecture Description Languages (ADL) [107], formulas and calculations concerning the system are examples of formalized knowledge. An ADL provides a clear and concise description of the architectural concepts used, which can be communicated, related, and reasoned about. The advantage of formalized knowledge is that it can be processed by computers.

Often, the different kinds of knowledge are used simultaneously. For example, despite that UML was not invented for it, UML is often used to model certain architectural concepts [29]. The model structure of UML contains formalized knowledge, which needs explanation in the form of documented knowledge. However, the use of the models is not unambiguous, and it is often found that UML is used in different ways. This implies the use of tacit knowledge to be able to understand and interpret the UML models in different contexts.

3.2.3 Problems in software architecture

There are several major problems with software architecture design [21, 76, 93]. These problems come from the large amount of tacit architectural knowledge. Currently, none of the existing approaches to describe software architectures (see section 3.2.2) give guidelines for describing the design decisions underlying the architecture. Consequently, design decisions only exist in the heads of the designers, which leads to the following problems:

- **Design decisions are cross cutting and intertwined.** Typical design decisions affect multiple parts of the design. However, these design decisions are not explicitly represented in the architecture. So, the associated architectural knowledge is fragmented across various parts of the design, making it hard to find and change the decisions.
- **Design rules and constraints are violated.** During the evolution of the system, designers can easily violate design rules and constraints arising from previously taken design decisions. Violations of these rules and constraints lead to architectural drift [119], and its associated problems (e.g. increased maintenance costs).
- **Obsolete design decisions are not removed.** When obsolete design decisions are not removed, the system has the tendency to erode more rapidly. In the current design practice removing design decisions is avoided, because of the effort needed, and the unexpected effects this removal can have on the system.

As a result of these problems, the developed systems have a *high cost of change*, and they tend to *erode quickly*. Also, the *reusability* of the architectural artifacts is *limited* if design decision knowledge vaporizes into the design. These problems are caused by the focus in the software architecture design process on the resulting artifacts (e.g. components and connectors), instead of the decisions that lead to them. Clearly, design decisions currently lack a first-class representation in software architecture designs.

3.3 Rationale in software architecture

To tackle the problems described in the previous section, the use of rationale is often proposed. Rationale in the context of architectures describes and explains the concepts used, alternatives considered, and structures of systems [70]. This section describes the use of rationale in software architectures. First, an abstract

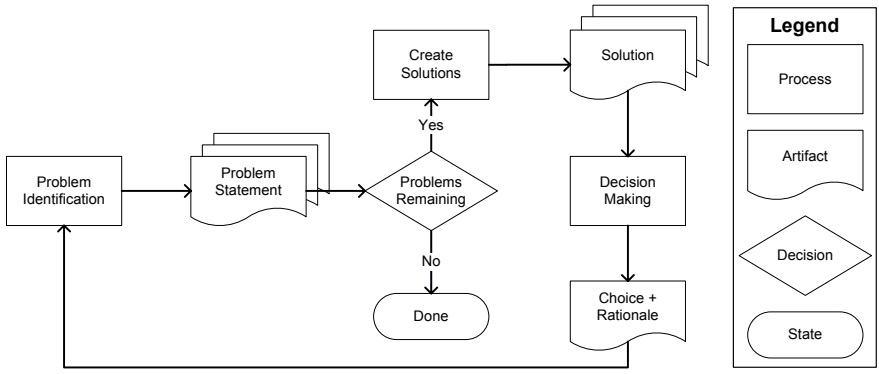

Figure 3.2: An abstract view on the rationale management process

rationale construction process is introduced in subsection 3.3.1. Then, the reasons for rationale use in software architecture are described in subsection 3.3.2. The section is concluded with a summary of problems for current rationale use in software architecture.

3.3.1 The rationale construction process

A general process for creating rationale is visualized in figure 3.2. First, the problems are identified (problem identification) and described in a problem statement. Then, the problems are evaluated (problems remaining) one by one, and solutions are created (create solutions) for a problem. These solutions are evaluated and weighed for their suitability of solving the problem at hand (decision making). The best solution (for that situation) is chosen, and the choice is documented together with its rationale (Choice + Rationale). If new problems emerge from the decision made, they have to be written down and be solved within the same process.

This process is a generalized view from different rationale based approaches (like the ones described in [42]). Take for example QOC, and the scenario described in [102]. The design of a scroll bar for a user interface is discussed. There are several questions (problems), like "Q1: How to display?". For this question, there are two options (solutions) described, “O1: permanent” and “O2: appearing”. In the described example, the second option is considered as the best one, and selected. However, this option generates a new question (problem), “Q2: How to make it appear?”. This new question needs to be solved in the same way. Other rationale management methods can be mapped on this process view too.

3.3.2 Reasons for using rationale in software architecture

As is discussed in [42], there are many reasons for using rationale in software projects. Here, the most important reasons are discussed, and related to the problems existing in software architecture.

- **Supporting reuse and change** During the evolution of a system and its architecture, the rules and constraints from previous decisions are often violated. Rationale needs to be used to give the architects insight into previous decisions.
- **Improving quality** As posed in the previous section, design decisions tend to get cross-cut and intertwined. Rationale based solutions are used to check consistency between decisions. This helps in managing the cross-cussing concerns.
- **Supporting knowledge transfer** When using rationale for communication of the design transfer of knowledge can be done over two dimensions: location (different departments or companies across the world) and time (evolution, maintenance). Transferring knowledge is one of the most important goals of an architecture.

3.3.3 Problems of rationale use in software architecture

As described in this section, rationale could be beneficial in architecture design. However, most methods developed for capturing rationale in architecture design suffer from the following problems:

- *Capture overhead*. Despite the attempt to automate the rationale capture process, both during and after the design, it is still a laborious process.
- For the designers, it is *hard to see the clear benefit* of documenting rationale about the architecture. Usually, most of the rationale captured is not used by the designer itself, and therefore capturing rationale is generally seen as boring and useless work.
- The rationale typically *loses the context* in which it was created. When rationale is communicated in documented or formalized form, additional tacit information about the context is lost.
- There is *no clear connection from the architectural artifacts to the rationale*. Because the rationale and the architectural artifacts are usually kept separated, it is very hard to keep them synchronized. Especially when the system is evolving, the design artifacts are updated, while the rationale documentation tends to deteriorate.

As a consequence of these problems, rationale based approaches are not often used in architecture design. However, as described in section 3.2.3, there is a need for documenting the reasons behind the design. The following section describes an approach which couples rationale to architecture.

3.4 Design decisions: the bridge between rationale and architecture

The problems from 3.2.3 and 3.3.3 can be addressed by the same solution. This is done by including rationale and architectural artifacts into one concept: the design decision. In the following subsection, the two processes from 3.2.1 and 3.3.1 are compared. In subsection 3.4.2, design decisions are introduced by example and a definition is presented in 3.4.3. The last subsection discusses designing with design decisions.

3.4.1 Enriching architecture with rationale

The processes described in subsections 3.2.1 and 3.3.1 have some clear resemblances. Problems (requirements) are handled by Solutions (software architectures / modifications), and the assessment determines if all the problems are solved adequately. The artifacts created in both processes tend to describe the same things (see figure 3.3). However, the software architecture design process focuses on the results of the decision process, while the rationale management focuses on the path to the decision.

Some knowledge which is captured in the rationale management process is missing in the architecture design process (depicted as black boxes in figure 3.3). There are two artifacts which contain knowledge that is not available in the software architecture artifact: not selected solutions and choice + rationale. On the other hand, the results of the design process (the architecture and architectural modifications), are missing in the rationale management process.

The concept of first-class represented design decisions, composed of rationale, architectural modifications, and alternatives, is used to bring the two processes together. A software architecture design process no longer results in a static design description of a system, but in a set of design decisions leading up to the system. The design decisions reflect the rationale used for the decision making process, and form the natural bridge between rationale and the resulting architecture.

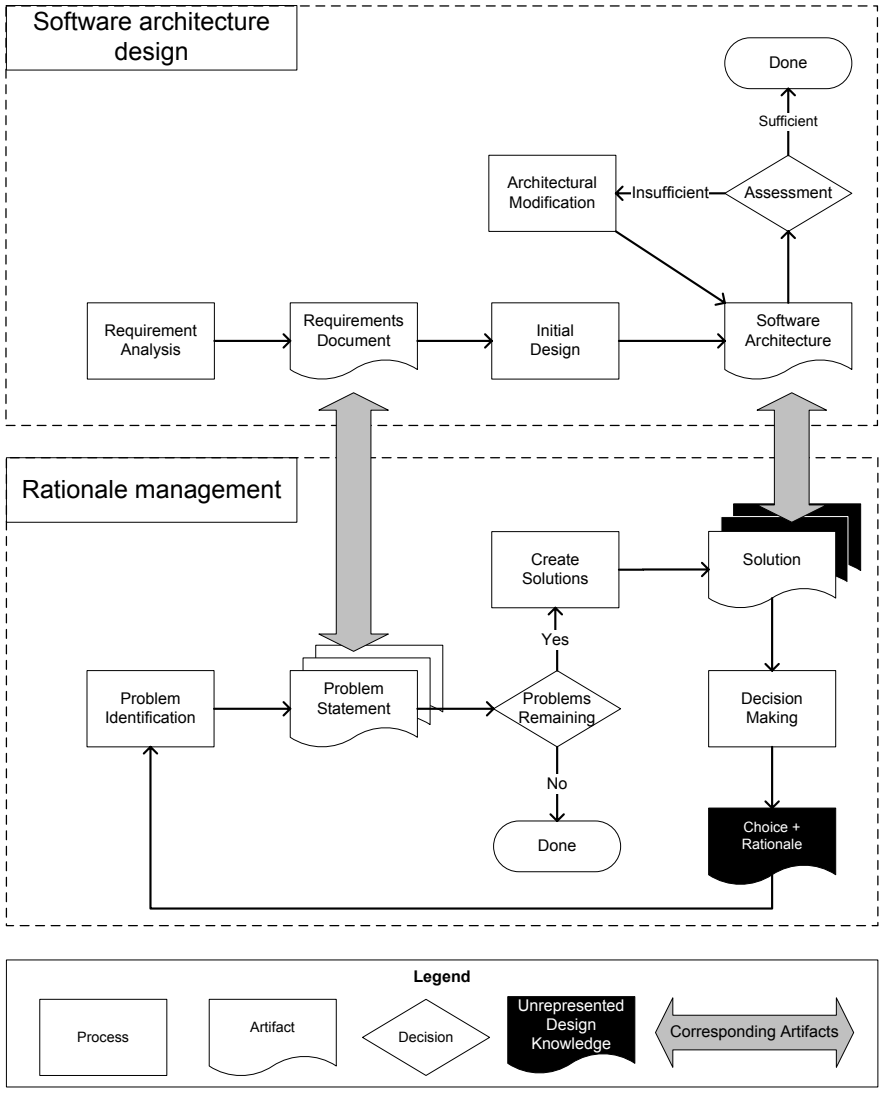

Figure 3.3: Similarities between software architecture design process and the rationale management process

3.4.2 CD player: a Design Decision Example

This subsection presents a simple case, which shows the impact of designing an architecture with design decisions. The example is based on the design of a compact disc (CD) player. Changing customers' needs have made the software architecture of the CD player insufficient. Consequently, the architecture needs to evolve.

The software architecture of the CD player is presented in the top of figure 3.4, the current design. The design decisions leading to the current design are not shown in figure 3.4 and are instead represented as one design decision.

The CD player's architecture is visualized in a component and connector view [29]. The components are the principal computational elements that execute at run-time in the CD player. The connectors represent which component has a run-time pathway of interaction with another component.

Two functional additions to the software architecture are described. First, a software-update mechanism is added. This is used to update the CD player, to make it easier to fix bugs and add new functionality in the future. Second, the internet connection is used to download song information for the played CD, like lyrics, additional artist information, etc.

As shown in figure 3.4, design decisions are taken to add the described functionality. The design decisions contain the rationale and the functional solution, represented as documentation and an architectural component and connector view. Note, that the rationale in the picture is shortened very much because of space limitations. The added functionality is directly represented by two design decisions, *Updater* and *SongDatabase*.

The first idea for solving the internet connectivity was to add a component which handled the communication to the Patcher. This idea was rejected, and another alternative was considered, to create a change to the Hardware Controller. This change enabled the internet connectivity for the Internet song db too, and was considered better because it could reuse a lot of the functionality of the existing Hardware Controller. Note that the view on the current design shows a complete architecture, while it is also a set of design decisions. The resulting design (figure 3.5) is visualized with the two design decisions taken: the *Updater* and the *SongDatabase*.

3.4.3 Design decisions

In the example of section 3.4.2, the software architecture of the CD player is the set of design decisions leading to a particular design, as depicted in 3.4. In the classical

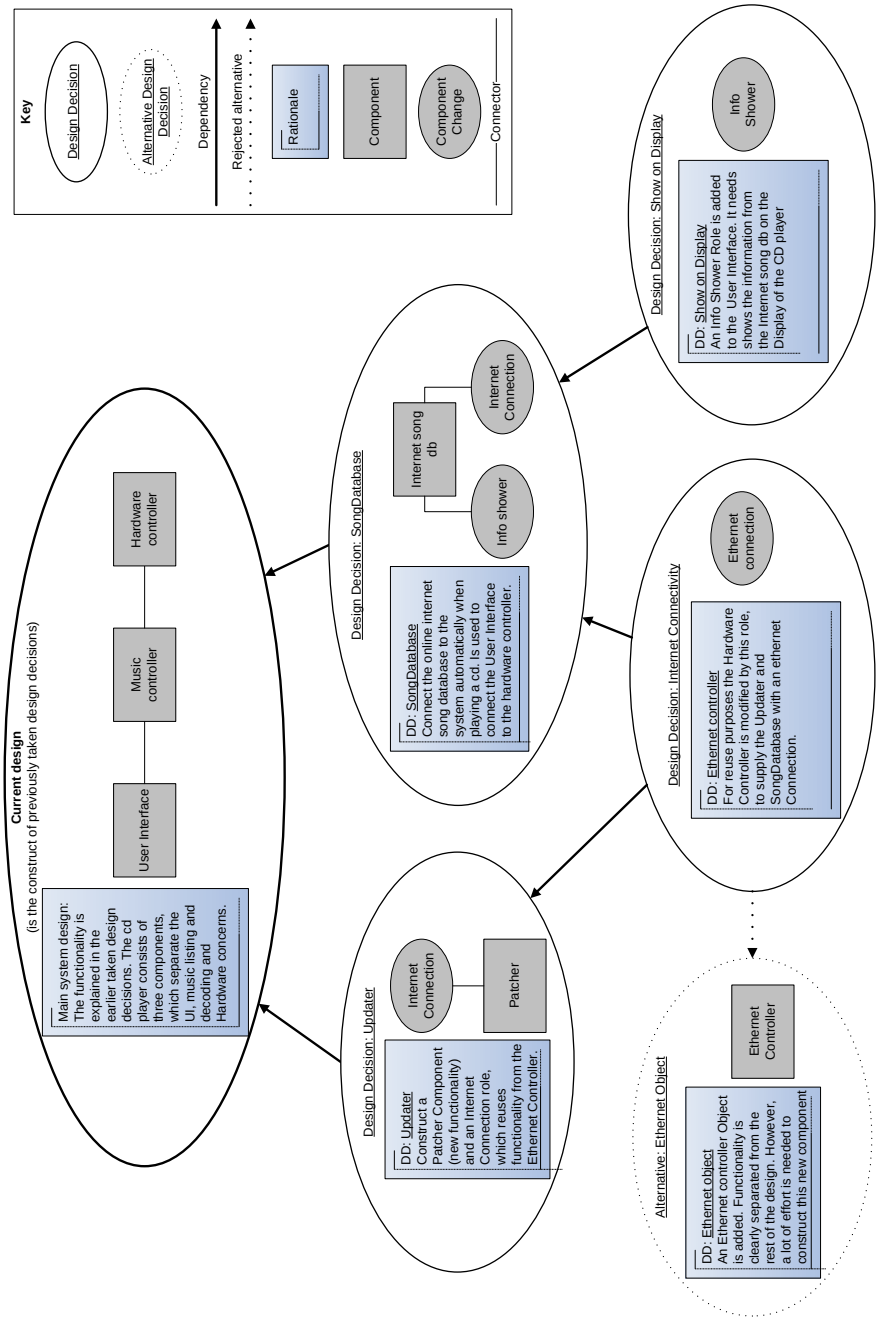

Figure 3.4: The architecture of a CD player with extended functionality

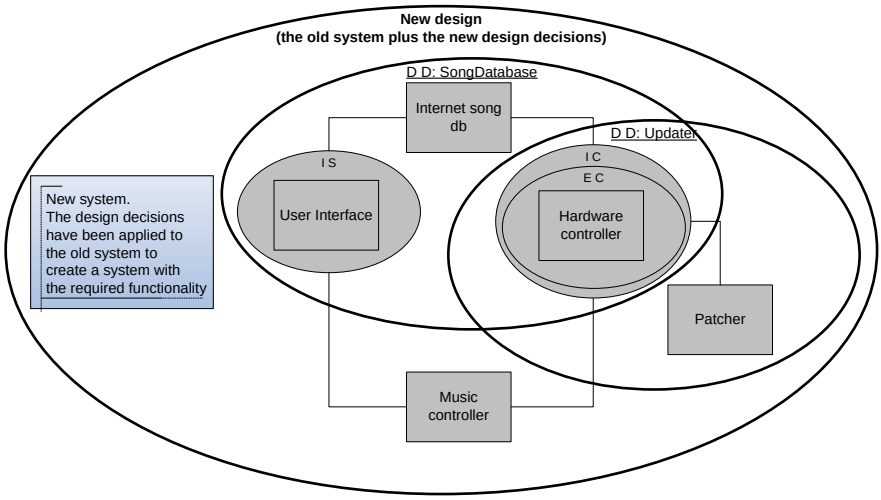

Figure 3.5: The result of the design decisions of figure 3.4

notion of system design only the result depicted in figure 3.5 is visible while not capturing the design decisions leading up to a particular design.

Although the term architectural design decision is often used [11, 29, 67], a precise definition is hard to find. Therefore, we define an architectural design decision as:

> A description of the choice and considered alternatives that (partially) realize one or more requirements. Alternatives consist of a set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements.

We detail this definition by describing the used elements:

- The *considered alternatives* are potential solutions to the requirement the design decision addresses. The *choice* is the decision part of an architectural design decision; it selects one of the considered alternatives. For example, figure 3.4 contains two considered alternatives for the connectivity design decisions. The Ethernet Object alternative is not selected. Instead, the Internet Connectivity is selected.

- The *architectural additions, subtractions*, and *modifications* are the changes to the given architecture that the design decision makes. For example, in figure 3.4 the Song Database design decision has one addition in the form of a

new component (the Internet Song Database), and introduces two modifications to components (info shower and internet connection).

- The *rationale* represents the reasons behind an architectural design decision. In figure 3.4, the rationale is briefly described within the design decisions.
- The *design rules* and *constraints* are prescriptions for further design decisions. As an example of a rule, consider a design decision that is taken to use an object-oriented database. All components and objects that require persistence need to support the interface demanded by this database management system, which is a rule. However, this design decision may require that the complete state of the system is saved in this object-oriented database, which is a constraint.
- Timely fulfillment of *requirements* drives the design decision process. The requirements not only include the current requirements, but also include requirements expected in the future. They can be either explicit, e.g. mentioned in a requirements document, or implicit.
- A design decision may result in *additional requirements* to be satisfied by the architecture. Once a design decision is taken, new insights can lead to previous undiscovered requirements. For instance, the design decision to use the Internet as an interface to a system will cause security requirements like logins, secure transfer etc.

The given *architecture* is a set of earlier made design decisions, which represent the architectural design at the moment the design decision is taken.

Architecture design decisions may be concerned with the application domain of the system, the architectural styles and patterns used in the system, COTS components and other infrastructure selections as well as other aspects described in classical architecture design. Consequently, architectural design decisions can have many different levels of abstraction. Furthermore, they involve a wide range of issues, from pure technical ones to organizational, business, political, and social ones.

3.4.4 Designing with design decisions

Existing design methods (e.g. [11] and [19]) describe ways in which alternatives are elicited and trade-offs are made. An architect designing with design decisions still uses these design methods. The main difference lies in the awareness of the architect, to explicitly capture the design decisions made and the associated design knowledge.

Section 3.2.3 presented key problems in software architecture. Designing with design decisions helps in handling these problems in the following way:

- **Design decisions are cross cutting and intertwined.** When designing with design decisions the architect explicitly defines design decisions, and the relationships between them. The architect is made aware of the cross cutting and intertwining of design decisions. In the short term, if the identified intertwining and cross cutting is not desirable, the involved design decisions can be reevaluated and alternative solutions can be considered before the design is further developed. In the long term, the architect can (re)learn which design decisions are closely intertwined with each other and what kind of problems are associated with this.
- **Design rules and constraints are violated.** Design decisions explicitly contain knowledge about the rules and constraints they impose on the architecture. Adequate tool support can make the architect aware about these rules and constraints and provide their associated rationale. This is mostly a long term benefit to the architect, as this knowledge is often forgotten and no longer available during evolution or maintenance of the system.
- **Obsolete design decisions are not removed.** In evolution and maintenance, explicit design decisions enable identification and removal of obsolete design decisions. The architect can predict the impact of the decision and the effort required for removal.

Designing with design decisions requires more effort from the architect, as the design decisions have to be documented along with their rationale. In traditional design, the architect forms the bridge between architecture and rationale. In designing with design decisions, this role is partially taken up by the design decisions.

Capturing the rationale of design decisions is a resource intensive process [42]. To minimize the capture overhead, close integration between software architecture design, rationale, and design decisions is required. The following section presents an example of an approach that demonstrates this close integration.

3.5 Archium

The previous section presented a general notion of architectural design decisions. In this section, a concrete example realization of this notion is presented: Archium [77]. First, the basic concepts of Archium are presented, after which this approach is illustrated with an example.

3.5.1 Basic concepts of Archium

Archium is an extension of Java, consisting of a compiler and run-time platform. Archium consists of three different elements that are integrated with each other. The first element is the architectural model, which formally defines the software architecture using ADL concepts [107]. Second, Archium incorporates a decision model, which models design decisions along with its rationale. Third, Archium includes a composition model, which describes how the different concepts are composed together.

The focus in this subsection is on the design decision model. For the composition and architectural model see [77]. The decision model (see figure 3.6) uses an issue-based approach [101]. The issues are problems that the solutions of the architectural design decisions (partially) solve. The rationale part of the decision model focuses on *design decision rationale* and not *design rationale* in general (see the 'DRL' section in chapter 1 of [42]).

Archium captures rationale in customizable rationale elements. They are described in natural text within the scope of a design decision. Rationale elements can explicitly refer to elements within this context, therefore creating a close relationship between rationale and design elements.

The motivation and cause elements provide rationale about the problem. The choice element chooses the right solution and makes a trade-off between the solutions. The choice results in an architectural modification.

To realize the chosen solution in an architectural design decision, the components and connectors of the architectural model can be altered. In this process, new elements might be required and existing elements of the design might be modified or removed. The architectural modification describes these changes, and thereby the history of the design. These architectural modifications are explicitly part of design decisions, which are first-class entities in Archium. This makes Archium capable of describing a software architecture as a set of design decisions [77].

Rationale acquisition (see chapter 1 of [42]) is a manual task in Archium. The approach tries to minimize the intrusiveness of the capturing process by letting the rationale elements of the design decisions be optional. The only intrusive factor is the identification and naming of design decisions.

The rationale elements are to a certain extend similar to that of DRL [101]. The *Problem* element is comparable to a *Decision Problem* in DRL. A *Solution* solves a *Problem*, likewise *Alternatives* do in DRL. The *Motivation* element gives more rationale about the *Problem* and is comparable to a supportive *Claim* in DRL. A

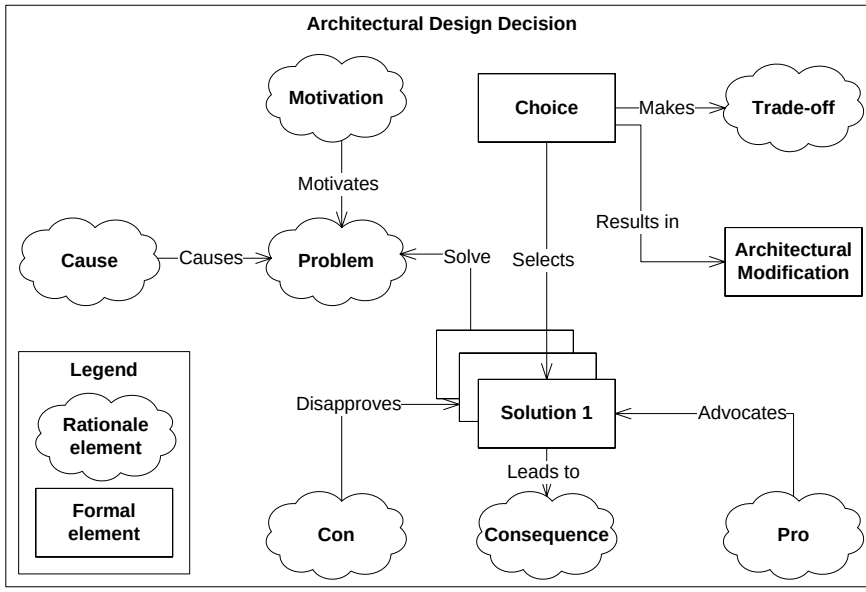

Figure 3.6: The Archium design decision model

Cause can be seen as a special instance of a *Goal* in DRL. The *Consequence* element is like a DRL *Claim* about the expected impact of a *Solution*. The *Pro* and *Con* elements are comparable to supporting and denying DRL *Claims* of a *Solution* (i.e. a DRL *Alternative*).

3.5.2 Example in Archium

An example of a design decision and the associated rationale in Archium is presented in figure 3.7. It describes the *Updater* design decision of figure 3.4. Rationale elements in Archium start with an @, which expresses rationale in natural text. In the rationale, any design element or requirement in the scope of the design decision can be referred to using square brackets (e.g. [iuc:patcher]). In this way, Archium allows architects to relate their rationale with their design in a natural way.

A design decision can contain multiple solutions. Each solution has a realization part, which contains programming code that realizes the solution. A realization can use other design decisions or change existing components. In the *InternetUpdate solution* the realization contains the *InternetUpdateChange*, which defines the *Patcher* component and the component modifications for the *Internet Connection* (see figure 3.4). The *IUCMapping* defines how the *InternetUpdateChange* is mapped onto the current *design*, which is an argument of the design decision.

```
design decision Updater(CurrentDesign design) {
  @problem {# The CD player should be updatable.[R4] #}
  @motivation {# The system can have unexpected bugs or require
                 additional functionality once it is deployed. #}
  @cause {# Currently this functionality is not present in the
            [design], as the market did not require this functionality
             before. #}
  @context {# The original [design]. #}

  potential solutions {
    solution InternetUpdate {
      architectural entities {
        InternetUpdateChange iuc;
        IUCMapping iucMapping;
      }
      @description {# The system updates itself using a patch, which
                      is downloaded from the internet. #}
      realization {
        iuc = new InternetUpdateChange();
        iucMapping = new IUCMapping(design,iuc);
        return design composed with iuc using iucMapping;
      }
      @design rules {# When the [iuc:patcher] fails to update, the
                       system needs to revert back to the previous state. #}
      @design constraints {# #}
      @consequences {# The solution requires the system to have a
                       [iuc:internetConnection] to work. #}
      pros {
        @pro {# Distribution of new patches is cheap, easy, and fast #}
      }
      cons {
        @con {# The solution requires a connection to the internet to
                work. #}
      }
    }
  /* Other alternative solutions can be defined here */
  }
  choice {
    choice InternetUpdate;
    @tradeoff {# No economical other alternatives exist #}
  }
}
```

Figure 3.7: The Updater design decision in Archium

To summarize, the architectural design decisions contain specific rationale elements of the architecture, thereby not only describing how the architecture has become what it is, but also the reasons behind the architecture. Consequently, design decisions can be used as a bridge between the software architecture and its rationale. The Archium environment shows that it is feasible to create architectures with design decisions.

3.6 Related work and further developments

This section describes related and future work. The related work focuses on software architecture, as the related work about rationale management is explained in

more depth in previous chapters of this book. After this, subsection 3.6.2 describes future work on design decisions.

3.6.1 Related work

Software architecture design methods [11, 19] focus on describing how the right design decisions can be made, as opposed to our approach which focuses on capturing these design decisions. Assessment methods, like ATAM [11], asses the quality attributes of a software architecture, and the outcome of such an assessment steers the direction of the design decision process.

Software documentation approaches [29, 67] provide guidelines for the documentation of software architectures. However, these approaches do not explicitly capture the way to and the reasons behind the software architecture.

Architectural Description Languages (ADLs) [107] do not capture the road leading up to the design either. An exception is formed by the architectural change management tool Mae [132, 161], which tracks changes of elements in an architectural model using a revision management system. However, this approach lacks the notion of design decisions and does not capture considered alternatives or rationale about the design.

Architectural styles and patterns [138] describe common (collections of) architectural design decisions, with known benefits and drawbacks. Tactics [11] are strategies for design decision making. They provide clues and hints about what kind of design decisions can help in certain situations. However, they do not provide a complete design decision perspective.

Currently, there is more attention in the software architecture community for the decisions behind the architectural design. Kruchten [93], stresses the importance of design decisions, and creates classifications of design decisions and the relationship between them. Tyree and Akerman [158] provides a first approach on documenting design decisions for software architectures. Both approaches model design decisions separately and do not integrate them with design. Closely related to this is the work of Lago [99], who models assumptions on which design decisions are often based, but not the design decisions themselves.

Integration of rationale with the design is also done in the design rationale field. With the SEURAT [24] system, rationale can be maintained in a RationaleExplorer, which is loosely coupled to the source code. This rationale has to be added to the design tool, to let the rationale of the architecture and the implementation be maintained correctly. DRPG [9] couples rationale of well-known design patterns

with elements in a Java implementation. Likewise SEURAT, DRPG also depends on the fact that the rationale of the design patterns is added to the system in advance.

3.6.2 Future work

The notion of design decisions as first-class entities in a software architecture design raises a couple of research issues. Rationale capture is very expensive, so how can we determine which design decisions are economically worth capturing? So far, we have assumed that all the design decisions can be captured. In practice, this would often not be possible or feasible. How do we deal with the completeness and uncertainty of design decisions? How can we support addition, change, and removal of design decisions during evolution?

First, design decisions need to be adapted into commonly used design processes. Based on this, design decisions can be formalized and categorized. This will result in a thorough analysis of the types of design decisions. Also, dependencies need to be described between the requirements and design decisions, between the implementation and design decisions and between design decisions among themselves.

Experiments by others have already proven that rationale management helps in improving maintenance tasks. Whether the desired effects outweigh the costs of rationale capturing is still largely unproven. The fact that most of the benefits of design decisions will be measurable after a longer period when maintenance and evolution takes place complicates the validation process. We are currently working on a case study which focuses on a sequence of architectural design decisions taken during evolution. Additional industrial studies in different domains are planned in the context of an ongoing industrial research project, which will address some of the aforementioned questions.

3.7 Summary

This chapter presented the position of rationale management in software architecture design. Rationale is widely accepted as an important part of a software architecture. However, no strict guidelines or methods exist to structure this rationale. This leaves the rationale management task in the hands of the individual software architect, which makes it hard to reuse and communicate this knowledge. Furthermore, rationale is typically kept separate from architectural artifacts. This makes it hard to see the benefit of rationale and maintaining it.

Giving design decisions a first-class representation in the architectural design creates the possibility to include problems, their solutions, and the rationale of these decisions into one unified concept. This chapter described an approach in which decisions behind the architecture are seen as the new building blocks of the architecture. A first step is made by the Archium approach, which illustrated that designing an architecture with design decisions is possible. In the future, we think that rationale and architecture will be used together in design decision-like concepts, bridging the gap between the rationale and the architecture.

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.

"Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled."

– Eoin Woods [145]