A Pattern-based Approach Against Architectural Knowledge Vaporization

Uwe van Heesch
University of Groningen,
Fontys University of Applied Sciences
Venlo, The Netherlands
u.vanheesch@fontys.nl

Paris Avgeriou
University of Groningen
Groningen, The Netherlands
paris@cs.rug.nl

Abstract

Architectural documentation is often considered as a tedious and resource intensive task, that is usually skipped or performed inadequately. As a result the rationale of the architect's decisions gets lost. This problem is known as architectural knowledge vaporization. We propose a documentation approach for architectural decisions concerning the application of software patterns. Based on the assumption that patterns and pattern languages incorporate generic architectural knowledge, we recommend reusing this documented knowledge in application specific architecture documentation to preserve an important part of the rationale, that went into the architect's decisions, while saving time compared to other documentation approaches.

1. Introduction

The documentation of architectural knowledge (AK) in software development often consists of design models and other artifacts recording the outcome of a design process, if at all, but misses the design decisions, the context and the rationale of a specific solution [11]. This is partially because comprehensive documentation is considered a resource-intensive task that interrupts the natural design flow and does not have an immediate benefit for the running project. By skipping it, some of the important details that a decision is based on, such as the decision context, assumptions, decision drivers, consequences and considered alternatives, get lost. This phenomenon is known as architectural knowledge vaporization [2, 15, 9]. It becomes particularly problematic when software systems have to be maintained and adjusted to changing requirements or execution contexts, because we often fail to understand the architect's original motivation for a specific design construct. Consequently design rules and constraints may be violated during system evolution [9].

AK is multi-faceted. [12] makes the distinction between application-generic and application-specific knowledge. The former consists of expertise that can be generally used in many different software projects. Architecture patterns and styles are some examples. The latter concerns knowledge of a specific application, gained during its development or evolution. Application-specific knowledge includes all decisions made during the architecting process as well as information about the problem and solution space of a concrete project. In this paper we will primarily address the vaporization of application-specific AK.

Additionally application-specific knowledge is further subdivided into context knowledge, reasoning knowledge and design knowledge according to [14]. The first involves information about the problem space, for example architecturally relevant requirements, forces and the problem context. Reasoning knowledge incorporates the rationale behind decisions including considered alternatives and trade-offs. Finally design knowledge is a collection of system designs like component and connector models or other architectural models. In [14] one more category of AK is identified, namely general knowledge which is the same as the aforementioned application-generic knowledge.

Architectural knowledge vaporization occurs with all three kinds of application-specific AK. For instance, although context knowledge is represented in requirement documents, specific requirements are seldom explicitly related to architectural decisions. It remains unclear which part of the problem is solved by an architectural decision. Reasoning knowledge is usually neither represented in requirements, nor in design documents. The rationale behind design decisions, including the

design alternatives, trade-offs, assumptions and consequences gets lost if it is not explicitly documented. Design knowledge is normally represented in design documents, but the latter are often incomplete, out of date or incorrect.

In this paper, we propose a lightweight approach to support software architects in systematically documenting application-specific architectural knowledge by reusing existing, codified application-generic AK encapsulated in software architecture patterns[1]. We focus on recording decisions made when applying patterns and make the concrete drivers that motivated the decisions explicit, by reusing the different types of AK that patterns incorporate. This allows to uncover alternatives, trade-offs and consequences.

The remainder of this paper is organized as follows: In section 2 patterns and pattern languages are discussed with respect to their architectural knowledge aspects. Section 3 presents a conceptual model of application-generic and application-specific architectural knowledge that supports our documentation approach. In section 4 we explain, how our approach can be used to document and analyze application-specific AK. Section 5 presents an example based on a pattern story that describes a real world architecting project. Section 6 derives some basic use cases for tool support and finally, in section 7 we conclude and give an outlook to future work.

2. Patterns and Pattern Languages

Patterns describe the solution to a problem in a certain context [1, 6]. Therefore they capture reusable knowledge providing a design to a specific, recurring design problem arising in a particular context and domain [13]. Architectural patterns capture knowledge that is not specific to a certain project, but application-generic. They contain information about the context and the problem space in terms of forces, as well as concrete implementation advice for the solutions they propose. Finally pattern descriptions reason about design rationale, alternatives, consequences and trade-offs that are performed when applying them. This knowledge is applicable in infinite cases. When the pattern is applied in a project and all its details are finalized, the pattern's knowledge becomes application-specific for that specific project.

A pattern language incorporates patterns from a particular domain or area of concerns and combines them to form a web of related patterns. It describes a whole system of patterns, including their interrelationships and dependencies. Patterns from a pattern language can be applied one after another in an incremental process of refinement to form a whole. Therefore each pattern defines its place in possible pattern sequences [16]. By concentrating more on the various relationships of patterns and the synergetic effects of their combination, pattern languages complement the AK captured in single patterns with additional knowledge of their relationships.

3. A Conceptual Model of Architectural Knowledge related to Architectural Patterns

To support the effective documentation and analysis of AK, we have developed a conceptual model that identifies the types of application-generic AK provided in the documentation of patterns and pattern languages and links this knowledge to application-specific AK. The model shown in figure 1 is divided in two parts: application-specific AK elements in the upper section, and generic AK elements in the lower section.

3.1 Application-generic Architectural Knowledge

The central concept in the application-generic section is architectural pattern. Patterns have a name and are applicable in a context. In addition, we provide a short description, the documentation date and the source of the pattern. One of the interesting relationships between patterns is the variant. Variants of patterns describe solutions to very similar, but different problems, that can vary in some of the forces [4]. The application of a variant potentially has different consequences. Variants are considered as independent patterns, with their own problem and solution descriptions.

The relationship type alternative means that patterns can be alternatively applied in a specific context. Pattern languages sometimes explicitly mention alternatives. The Layers pattern for instance can be seen as an alternative for the Microkernel pattern (both [4]) concerning the structural decomposition of a system.

Another type used to capture the relationship between patterns is combination. This type encapsulates all other interdependencies between patterns. Pattern combinations can be very rich and complex and many subtypes exist, that are not made explicit in our model. A simple example for the combination relationship type is the Model-View-Controller pattern [4] that

[1] In the remainder of this paper we will also refer to software architecture patterns as patterns.

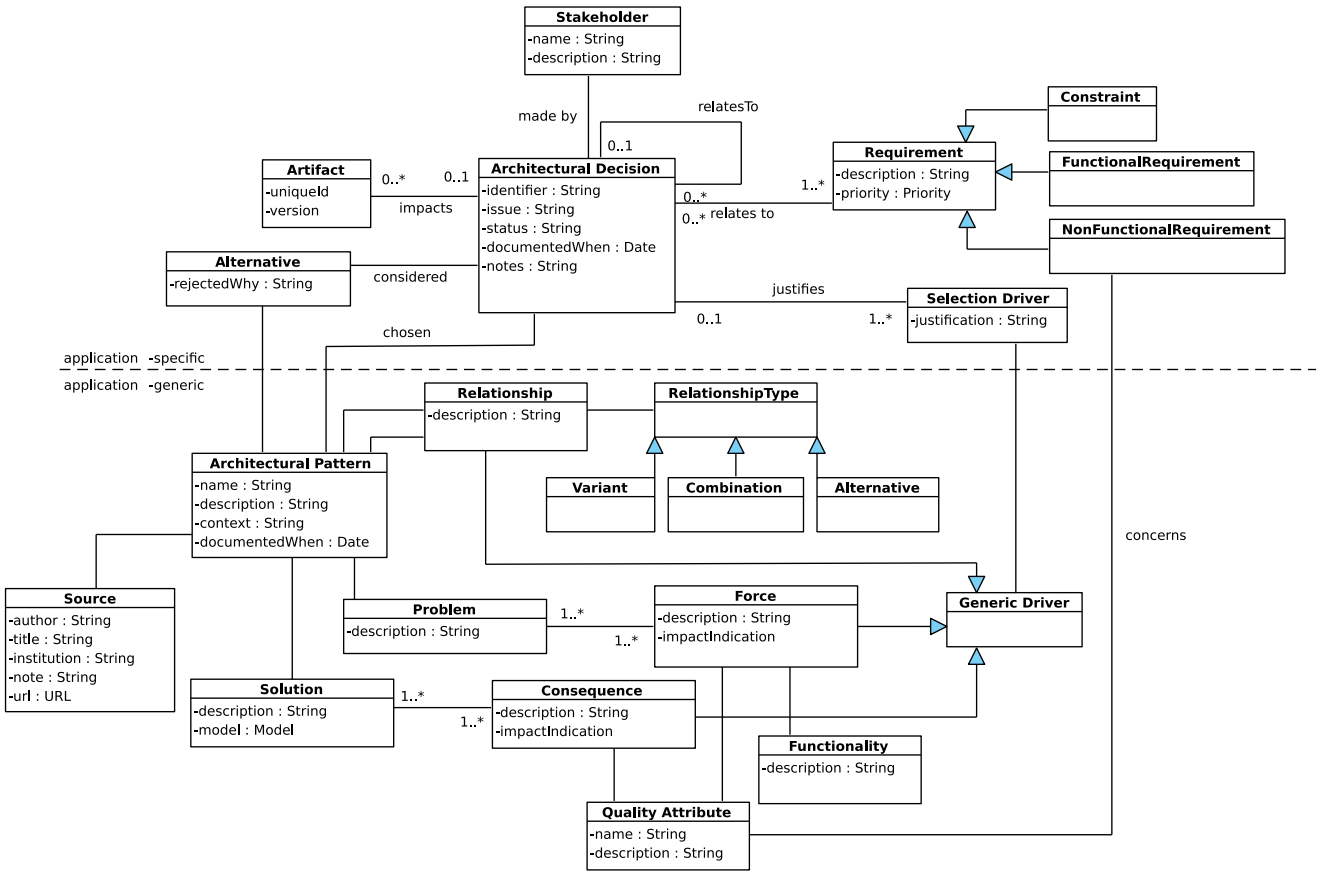

Figure 1. A conceptual model of AK related to Architectural Patterns

uses the Publisher-Subscriber pattern [4] for state change propagation. The Broker pattern [4] is another example. It can be combined with many other patterns to realize distributed systems.

The relationship between two patterns is a potential reason for applying a pattern. Consider a situation where multiple patterns have already been chosen and applied. When searching for additional patterns that address open requirements, it makes sense to choose a pattern that is known to work well with the previously selected ones, if multiple alternatives exist. By modelling the different types of relationships between patterns, we conserve reasoning knowledge that architects and stakeholders can revert to when analyzing architectural decisions later.

The context knowledge of patterns is described in terms of a problem consisting of multiple forces, while their design knowledge is incorporated in the representation of the solution. The application of architectural patterns has an impact on the quality attributes (QAs) of the target architecture by resolving related forces [10, 7, 16]. Therefore quality attributes are linked both to the forces and the consequences. The *impactIndication* field in the consequence and force classes can be used to express, whether a quality attribute is impacted positively or negatively. The terms *force* and *consequence* refer to the format for pattern descriptions used in [4]. Some pattern description formats explicitly describe forces and consequences; others like [5] do not mention them explicitly, but nevertheless contain the same information in other parts of the description. Forces and consequences are potential drivers for the selection of an architectural pattern, as they denote requirements and desired properties of the target architecture.

3.2 Application-specific Architectural Knowledge

The upper section of our model is concerned with the documentation of application-specific AK, and specifically architectural decisions. We are not aiming at capturing all decisions that need to be made in a software project, but those concerning the application of architectural patterns.

Patterns are abstract and generic (often called “half-baked”), and have to be finished off when they are applied in a project. We do not consider that the architecture of a system is composed only of patterns; however a fundamental part of the architecture can be constructed using architectural patterns. If so, then the AK encapsulated in the patterns is transformed into application-specific. For documenting the design decisions related to applying the patterns, the relevant parts of the pattern documentation can be reused.

In the AK model presented in figure 1, a decisions addresses an issue. This can be the specification of a concrete problem to be solved, or something that needs to be decided without directly solving a problem, for instance the choice of a programming language. A decision can be related to zero or more requirements. Besides functional and non-functional requirements, a constraint is also seen as a requirement that reduces the number of possible outcomes of a decision. A non-functional requirement concerns a specific quality attribute.

An architectural decision can also be related to other architectural decisions. As an example, if the shared repository pattern was chosen to share data between two subsystems, then a related decision could concern the choice of a database access strategy like the one provided in the Table Data Gateway pattern [5]. The concrete types of relationships between architectural decisions are subject to further research.

Decisions are personalized by also documenting the decision-maker and the date when the decision was made.

The notes in architectural decision should give hints on where the chosen pattern was applied in the architecture. The same pattern can be applied in different places for different reasons.

Often, architectural decisions concern the choice of a solution among alternatives. Therefore the model explicitly considers alternative patterns along with the reasons for their rejection.

The AK behind the decision is captured by referencing an architectural pattern along with the relevant drivers for the selection of the pattern. As mentioned earlier, forces, consequences and pattern relationships may be referenced as relevant drivers. By doing so, the rationale behind the decision that the architect made is recorded. The resulting documentation of the architecture not only explains which patterns were applied, but also why they were applied, by explaining which of all the potential drivers were decisive for the architect. This does not replace application-specific design and requirements documents. It is in the application-generic nature of patterns that they are abstract and have to be adjusted and modified to fit in a concrete design situation. Referencing their pure form in an application-specific context however has the benefit that the concept of the pattern is much clearer and easier to understand in this representation than the concrete design of the architecture after the pattern was applied. Nevertheless we also tried to find an effective way of documenting the real design outcome of a decision. We suppose that ADs affect many different documentation artifacts, for example UML component or class diagrams or textual documentation. Our assumption is also that the most software projects make use of versioning systems like Mercurial[2] or Subversion[3]. By documenting the version numbers of all relevant versioning repositories before and after an AD was enforced, we conserve the architectural delta. This is the part of the architecture that the architect actually changed to apply the chosen pattern, be it textual documentation, or UML diagrams or other artifacts.

[2] see http://mercurial.selenic.com

[3] see subversion.tigris.org

4. Documentation and Analysis of Architectural Knowledge

Our approach is complementary to existing architecting processes [8]. It can be used at the time when architects apply architectural patterns to satisfy architectural requirements. The patterns described in the generic part of the model should be managed in a repository that can be used from software development projects. We will describe some basic use cases for tool support in section 6.

4.1 Documentation of AK

Figure 2 shows the process of documenting architectural decisions concerning architectural patterns. The process starts when the architect decides to apply a pattern in the architecture. Subsequently the core of the decision (i.e. a unique identifier, the issue, the status, the documentation date) is documented, along with the stakeholder who made the decision. If applicable, related decisions and related requirements are linked to the current decision. Then a snapshot of all relevant project artifacts is taken. This can be done capturing version numbers of artifacts managed in versioning systems like Subversion or Mercurial for instance.

Every pattern that has been applied or considered in the software project should be documented in a pattern repository. This repository can be reused in many different software projects. Hence the patterns have to be added to the repository once, if they do not exist yet.

Once the pattern is chosen and eventually documented, the architect links the chosen pattern and the considered alternatives to the current decision. Finally the concrete drivers that lead him in choosing the pattern are also linked to the decision. The last aspect is very important. A pattern potentially has many benefits and liabilities expressed in terms of consequences but also in terms of forces. But perhaps not all of them were relevant when the architect decided to choose the pattern. This information is valuable when the decision is analyzed later, e.g. during the maintenance process. It clarifies *why* the architect chose the pattern.

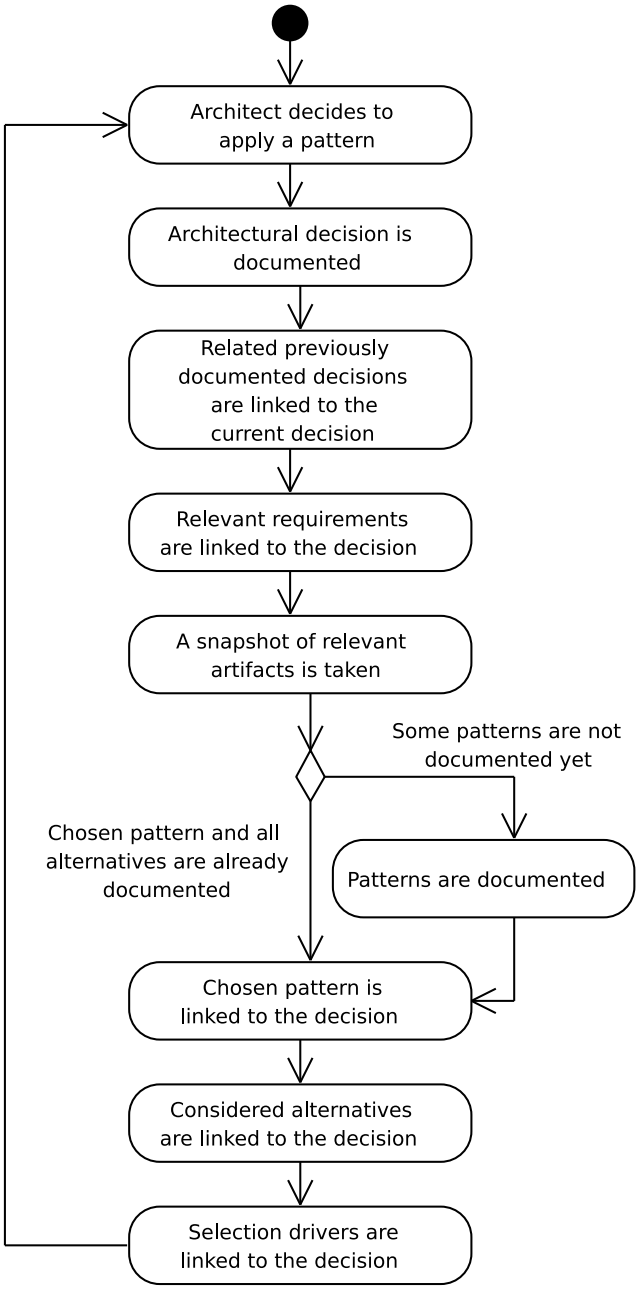

Figure 2. Documenting Architectural Decisions

4.2 Analysis of the documented AK

In this section we will explain how the different types of architectural knowledge map to the elements in the presented conceptual model, and consequently to the AK documented using our approach.

4.2.1 Context Knowledge

Context knowledge is spread across different parts of the conceptual model. The issue field, which is part of the AD, gives a first impression of the decision context. The relation to specific requirements explains which constraints, functional and non-functional requirements are addressed by the decision. Finally the forces and problem description of the related pattern can be browsed to understand the decision context.

4.2.2 Design Knowledge

We suppose that large parts of the design knowledge are documented in design artifacts like box-and-line or UML diagrams. The documented patterns, however, also contribute to understanding the design.

One the one hand, the design presented in the solution part of a pattern is generic and incomplete; it has to be adapted to fit in concrete design situations. On the other hand, the design documentation of a project is more concrete and complete than the design snippets presented in pattern descriptions; but it is also more complicated and harder to understand. The pattern solution presents a very clear view that is reduced to the essentials of the respective pattern and thus easier to understand. This clear view is complemented by the architectural delta, which can be gained by comparing the artifact snapshot taken before an architectural decision was enforced to a snapshot of the same artifact after it was enforced. This allows to analyze the influence of a decision on the system design and documentation. Stakeholders can use both, the clear pattern design and the actual realization in the architecture to understand the design behind an architectural decision. In section 6 we give two examples.

4.2.3 Reasoning Knowledge

Making reasoning knowledge explicit, particularly the rationale behind the decisions is the most challenging part. We make use of the concrete drivers that were documented along with the chosen patterns. The drivers explain why the architect chose a specific pattern. The considered alternatives, which were also documented, correspond to the 'paths not taken' during the decisions making process. Additionally, the pattern descriptions of the chosen and considered patterns contain valuable information about consequences and trade-offs that were made when applying the pattern.

5. An example

To exemplify our approach we will use parts of a pattern story describing the design of a real-world warehouse management system presented in [3]. We focus on the part of the story that addresses the architecture of the warehouse system. The story is suitable for explaining our approach, because it describes architectural decisions concerning the application of architectural patterns taken in a real software project. The complete presented set of decisions and patterns is taken from [3].

5.1 A Pattern story

The pattern story concerns a control system for warehouse management. The functional requirements cover stock management, order management, shipping, receiving, material flow control and the warehouse topology management.

The following non-functional requirements were identified by the authors:

- Distribution: The functionality of the system must be accessible by clients remotely.
- Performance: The system must ensure that all orders are performed efficiently without visible interruption.
- Scalability: The system must be scalable in the number of warehouse "bins" and computational devices connected to the warehouse.
- Availability: The system demands a minimum availability of 99.999%.
- Persistence: Most state information maintained by the warehouse system must be persistent.
- Portability: The warehouse must run on different hardware platforms and operating systems.
- Dynamic configuration: The system must be runtime configurable.

- Human-computer interaction: A wide variety of user interfaces must be supported.
- Component integration: The system must integrate useful third-party products.
- Generality: The system must provide a general solution that is usable in a variety of cases in the warehousing domain.

The listed system requirements are reduced to information needed to show the applicability of our approach. Please refer to [3] for detailed descriptions.

5.2 Architectural Decisions made in the story

The following architectural decisions made in the story concern architectural patterns: *Layers* (AD1), *Domain Object* (AD2), *Explicit Interface* and *Encapsulated Implementation* (AD3), *Broker* (AD4), *Model View Controller* (AD5), *Half-Object Plus Protocol* (AD6), *Actice Object* (AD7), logging Domain Object using *Leader/Followers* (AD8), *Database Access Layer* (AD9), *Component Configurator* (AD10).

To exemplify our approach we show its use in documenting AD1 and AD5. Then we will explain how to analyze the application-specific architectural knowledge behind the decisions. Remember that one of the goals of the presented approach is to keep the documentation effort for the architect low. The form of documentation is derived from the model presented in figure 1. Additional to the information in the tables presented in the following subsection the decision maker, status and time would have been documented in a real project, but this information was not available in the pattern story. Besides, according to the model, a reference to artifacts would exist that shows the resulting design in UML diagrams for instance. The design that resulted from the decisions here is taken from the pattern story as well.

We documented all patterns that are mentioned in the example according to our conceptual model as well, based on the descriptions in [4]. In this example we will only refer to excerpts of the documented patterns. We focus on the documentation of the actual architectural decisions.

5.2.1 AD1 - Layers

Architectural decision 1 is documented in Table 1. It concerns the usage of the Layers architectural pattern. The following parts of the application-specific architectural knowledge behind this decision can be made explicit using the pattern description.

- **Context knowledge**: The documented context refers to the partitioning of the warehouse system into coherent parts. The general context of the Layers pattern refers to the decomposition of a large system. The problem description of the Layers pattern in the generic part of the conceptual model leads to further insights. The Layers pattern is applicable if the system consists of a mix of high-level and low-level functionality, where the high-level operations rely on low level operations. This is the case here, high-level functionality such as order management and shipping rely on low-level functionality such as material flow control and warehouse topology management. This application-specific knowledge can be derived from the general description of the Layers pattern without explicitly being documented during the architecting process. It puts the decision in the right context. The architect was searching for a way to decompose the whole system into coherent parts while taking the specific characteristics of the warehouse system into account.
- **Design knowledge**: Although the information in table 1 does not contain any design information, a decent part of the design knowledge behind the decision can be inferred from the pattern description. First the general description of the Layers pattern includes a design template depicting the structure of the Layers pattern. It gives a clear view of the involved components and their connectors. The whole system is structured into multiple layers. Starting at the bottom, each layer contains components at the same level of abstraction. Services provided by one layer are used by components in the above layer. Note that this short description is very general in nature. It is derived from the solution documented along with the Layers pattern in out conceptual model and has no reference to the concrete case of the warehouse management system. It gives a first impression of the design that resulted from the architectural decision. More concrete information can be gathered from the reference to impacted artifacts that is part of our conceptual model. It is a pointer from the decision to the resulting design. Here, the system was actually partitioned into five layers. A presentation layer, a business process layer, a business object layer, an infrastructure layer and an access layer. In this example the information as well as any other information about the warehouse system is taken from the pattern story in [3]. There we can also find the components that reside in the respective layers.

Seq. No	1
Issue	The warehouse system needs to be partitioned into coherent parts.
Specific Requirements	• Portability • Generality
Arch. Pattern	Layers
Sel. Drivers	• D1 (Force): Complex components need further decomposition. • D2 (Force): Support changeability. • D3 (Force): Support grouping of components along responsibilities. • D4 (Force): Support task division between programmers. • D5 (Consequence): Support for standardization • D6 (Consequence): Dependencies between components are kept local • D7 (Consequence): Separation of concerns

Table 1. Architectural Decision 1

- **Reasoning knowledge**: Table 1 mentions a couple of forces and consequences that were decisive for choosing the Layers pattern. They indicate why the pattern was chosen. Note that there are other, unmentioned forces and consequences concerning the Layers pattern. One of the key advantages of the Layers pattern is support for reuse of system layers. This factor implicitly plays a role when applying the Layers pattern, but in this case it was not relevant for the architect. This information can help a lot in understanding the decision and its consequences. The application of the Layers pattern has some potential liabilities. It might lead to communication overhead when upper layers have to pass multiple intermediate layers to use functionality of low layers. This might lead to lower efficiency compared to a non-layered system where components may access each other freely. This is a trade-off the architect made when choosing the Layers pattern.

5.2.2 AD5 - Model-View-Controller

The documentation of architectural Decision 5 is shown in Table 2. It concerns the usage of the Model-View-Controller (MVC) Pattern.

- **Context knowledge**: The documented context refers to the separation of the warehouse's user interfaces from the warehouse functionality. The general context of the Model-View-Controller pattern taken from [4] is providing flexible user interfaces for interactive applications. The problem section of the MVC pattern includes more information.

Seq. No	5
Issue	The warehouse's User Interface needs to be separated
Specific Requirements	• Human-computer interaction • Generality
Arch. Pattern	Model-View-Controller
Sel. Drivers	• D1 (Force): The user interface must reflect data changes immediately. • D2 (Force): Support for changeability. • D3 (Force): The functional core of components needs to be separated from the user interface. • D4 (Consequence): Views are synchronized. • D5 (Consequence): System parts are exchangeable.

Table 2. Architectural Decision 5

User interfaces are likely to change more often than system functionality. Different users have different requirements regarding the user interface and often several different user interfaces must be incorporated. This also describes the specific problem the architect wanted to solve for the warehouse system by applying the MVC pattern.

- **Design knowledge**: The description of the MVC pattern explains its general design. The user interface is divided in a model containing the data, Views that display the information to the user and controllers that manage user input. Views and controllers act as observers of the model and are informed automatically if data in the model is updated. This is the application-generic solution that the MVC pattern proposes. Again, the artifact reference adds application-specific design knowledge. Here we could see, that the views and controllers reside in the presentation layer, while the model is represented in the business process layer. Change propagation components were introduced for providing messaging functionality that is used to inform controllers and views if an update occurs. Please refer to [3] for more details. This example shows, that the straightforward design that is proposed by the MVC pattern description helps to understand the application-specific design solution. In the real design the involved components have more than one functionality and names that refer to the respective application domain. This makes it harder to find out, that the MVC pattern has actually been applied and even harder to understand how the solution works.
- **Reasoning knowledge**: Table 2 shows the selection drivers that let the architect choose the Model-View-Controller (MVC) pattern [4]. They indicate why the pattern was chosen. There are other potential drivers for choosing MVC. For example the possibility for presenting the same information differently in multiple views. Although this possibility is automatically given when applying MVC it was not decisive for the architect. Some negative consequences come along with the MVC pattern. MVC introduces a very close coupling between view and controller. When porting the user interface it is very likely that both have to be changed. MVC also leads to more complexity and the potential for excessive updates of multiple views resulting in a large communication overhead. These are liabilities that the architect accepted for getting the advantages of the MVC pattern.

6. Tool support

To tap the full potential of the presented approach, comprehensive tool support is indispensable. We have started to elicit some basic high-level use cases for a tool supporting our approach.

The following two use cases support the management of the application-generic AK from the conceptual model in figure 1. Essentially, there has to be create, retrieve, update and delete (CRUD) functionality to manage all entities in this part of the model:

- UC1: **Manage architectural patterns**: This use case includes adding, updating and deleting architectural patterns. According to our conceptual model, every pattern must be described in terms of a context, a category, a problem statement, forces, a representation of the solution and consequences. Additionally technologies supporting the patterns can be managed.
- UC2: **Manage pattern relationships**: Maintain the various relationships between the patterns in the repository. Basic types of relationships are variant, combination and alternative. Each of them should be refineable.

To support the architectural documentation process presented in section 4, the following use cases are applicable:

- UC3: **Document architecture relevant requirements**: Functionality is needed to select and document architecture relevant requirements from existing project documentation.
- UC4: **Add architectural decision**: When the architect decides to apply a pattern in the architecture, it needs to be recorded as an architectural decision. Every documented decision references the relevant drivers for choosing the concerned pattern, as well as a set of requirements that the decision satisfies. Additionally it must be possible to link a decision to a repository version in versioning systems like CVS or Subversion. By this it is possible to document how existing design documents or generally all versioned project artifacts have been affected by the decision.
- UC5: **Explore decision rationale**: Additionally to the chosen pattern, the satisfied requirements and the relevant drivers, it must be possible to explore the consequences of the chosen pattern, alternative patterns, variants and other patterns, that are related to the chosen pattern. This information can be taken from the data captured in the application-generic part of the conceptual model. It should also be possible to visualize the change in design documents and other artifacts that the decision caused.

Some of the presented use cases reference existing project artifacts like requirement documents and versioning systems. Therefore it would make sense to develop the documentation tool as an extension to existing Computer Aided Software Engineering (CASE) tools that are used to create analysis and design documents. We are going to implement these use cases in prototypes to validate our concepts. One of our goals is to provide lightweight tooling, that supports our approach and easily integrates with existing CASE tool chains to keep the barriers for its usage low.

7. Conclusion and Outlook

In this paper we presented an approach to address architectural knowledge vaporization by documenting decisions concerning the application of architectural patterns. A part of application-specific architectural knowledge comes from instantiating application-generic architectural knowledge in the form of architectural patterns. We make use of this by referencing chosen patterns, considered alternatives and specific decision drivers when documenting architectural decisions. This keeps the documentation effort that an architect has to spend during the architecting phase low, while preserving great parts of the rationale that went into the decisions. The patterns that are referenced do not have to be documented during the architecting phase or by the architect himself. As they are general in nature, they can be easily documented in advance and then be reused in many different software projects. We proposed a conceptual model of AK that supports our approach.

The process of documenting ADs as well as the analysis of the specific knowledge can easily be supported by tools. We described some basic use cases for that.

Our approach needs to be extended. Patterns do not cover the whole problem space of applications. Not every architectural problem can be solved by applying a pattern. If no suitable pattern exists, then the current approach cannot be used. Additionally, not all architectural decisions concern the usage of architectural patterns. Some architectural decisions concern the selection of technologies instead of patterns. Existing software systems, frameworks, middlewares and application platforms

are some examples. These decisions cannot be documented using our model yet. However, we assume that technologies can be described similarly to architectural patterns. We are currently looking into documenting architectural decisions concerning the use of technologies in the same way as decisions concerning patterns.

One might argue that documenting patterns in a repository is actually a high effort that interrupts the architect's design flow, but this is not necessarily the case. First, the patterns in the repository do not have to be documented by the architect himself. There's not much expertise needed to add a pattern to the repository based on a pattern description in a book or an article. Second the patterns do not necessarily have to be documented during the architecting phase. It would be even better to have a repository of potentially applicable patterns before the architecting phase starts. Ideally such a repository would even be publicly available for use and contribution.

We proposed a documentation approach that allows architects to record architectural decisions related to patterns. By referencing chosen patterns, considered alternatives and the concrete drivers that induced the architect to choose the pattern, we implicitly preserve the rationale behind the decision including variants, consequences and related patterns. Effort has to be spent once to document the patterns. They can then be reused by referencing them in different software projects. The documentation of the decisions does not require extra effort.

8. Acknowledgement

Thanks to Neil Harrison for giving good advice and providing useful feedback during the shepherding of this paper for EuroPLoP 2009.

9. Copyright

References

[1] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series)*. Oxford University Press, August 1977.

[2] J. Bosch. Software architecture: The next step. *Software Architecture*, pages 194–199, 2004.

[3] F. Buschmann, K. Henney, and D. C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. Wiley, May 2007.

[4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley, August 1996.

[5] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.

[6] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, November 1994.

[7] N. Harrison and P. Avgeriou. Leveraging architecture patterns to satisfy quality attributes. In *Proceedings. First European Conference on Software Architecture*. Springer LNCS, 2007.

[8] C. Hofmeister, P. Kruchten, R. Nord, H. Obbink, A. Ran, and P. America. A general model of software architecture design derived from five industrial approaches. *Journal of Systems and Software*, 80(1):106–126, January 2007.

[9] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proceedings. WICSA 2005. 5th Working IEEE/IFIP Conference on Software Architecture, 2005*, pages 109–120, Washington, DC, USA, 2005. IEEE Computer Society.

[10] A. Jansen, J. Bosch, and P. Avgeriou. Documenting after the fact: Recovering architectural design decisions. *Journal of Systems and Software*, 81(4):536–557, April 2008.

[11] A. Jansen, J. van der Ven, P. Avgeriou, and D. K. Hammer. Tool support for architectural decisions. In *Proceedings. Working IEEE/IFIP Conference on Software Architecture, 2005*, volume 0, pages 4+, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[12] P. Lago and P. Avgeriou. First workshop on sharing and reusing architectural knowledge. *SIGSOFT Softw. Eng. Notes*, 31(5):32–36, 2006.

[13] D. C. Schmidt and F. Buschmann. Patterns, frameworks, and middleware: their synergistic relationships. In *Proceedings. 25th International Conference on Software Engineering, 2003.*, pages 694–704, May 2003.

[14] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. Ali Babar. A comparative study of architecture knowledge management tools. *Journal of Systems and Software*, September 2009.

[15] J. Ven, A. Jansen, J. Nijhuis, and J. Bosch. Design decisions: The bridge between rationale and architecture. In *Rationale Management in Software Engineering*, chapter 16, pages 329–348. 2006.

[16] U. Zdun. Systematic pattern selection using pattern language grammars and design space analysis. *Software: Practice and Experience*, 37, 2006.