

Chapter 2

Knowledge Management in Software Architecture: State of the Art

Rik Farenhorst and Remco C. de Boer

Abstract Architectural knowledge has played a role in discussions on design, reuse, and evolution for over a decade. Over the past few years, the term has significantly increased in popularity and attempts are being made to properly define what constitutes ‘architectural knowledge’. In this chapter, we discuss the state-of-the-art in architectural knowledge management. We describe four main views on architectural knowledge based on the results of a systematic literature review. Based on software architecture and knowledge management theory we define four main categories of architectural knowledge, and discuss four distinct philosophies on managing architectural knowledge, which have their origin in the aforementioned views. Whereas traditionally tools, methods, and methodologies for architecture knowledge management were confined to a single philosophy, a trend can be observed that state-of-the-art approaches take a more holistic stance and integrate different philosophies in a single architecture knowledge management approach.

2.1 Introduction

Over the past few years, the concept of ‘architectural knowledge’ has become more prominent in literature and attempts are being made to arrive at a proper definition for this concept. In this chapter we present the state-of-the-art in architecture knowledge management. To this end, we look at what precisely entails architectural knowledge and what predominant philosophies exist for managing such knowledge.

In answer to the question ‘what is architectural knowledge?’, in Sect. 2.2 we describe four main views on architectural knowledge that emerged from a systematic

Rik Farenhorst (✉)

VU University Amsterdam, The Netherlands e-mail: rik@cs.vu.nl

Remco C. de Boer

VU University Amsterdam, The Netherlands e-mail: remco@cs.vu.nl

literature review, and explore their commonalities and differences. Using two orthogonal architectural knowledge dimensions, we define four categories of architectural knowledge. The potential knowledge conversions between these categories, which we describe in Sect. 2.3, together form a descriptive framework with which different architecture knowledge management philosophies can be typified. This framework shows that the differences between the four views on architectural knowledge are very much related to the different philosophies they are based on.

Although there exist several single-philosophy approaches that have both their origin and scope on only one of the main philosophies, in recent years a shift towards more overarching approaches can be observed. Four of such trends for architecture knowledge management are discussed in Sect. 2.4.

2.2 What Is ‘Architectural Knowledge’?

To be able to understand what architectural knowledge entails, we have conducted a systematic literature review in which we explored the ‘*roots*’ architectural knowledge has in different software architecture communities. Details on the protocol followed in this systematic literature review can be found in Sect. 2.5. The review revealed four primary views on architectural knowledge. In Sect. 2.2.1 we elaborate upon these views, and in Sect. 2.2.2 we formulate a theory on architectural knowledge by looking at the commonalities and differences between these views.

2.2.1 Different Views on Architectural Knowledge

Architectural knowledge is related to such various topics as architecture evolution, service oriented architectures, product line engineering, enterprise architecture, and program understanding, to name but a few. In the literature, however, there are four main views on the use and importance of architectural knowledge. Those four views – pattern-centric, dynamism-centric, requirements-centric, and decision-centric – are introduced in this section.

2.2.1.1 Pattern-Centric View

In the mid-nineties, patterns became popular as a way to capture and reuse design knowledge. People were disappointed by the lack of ability of (object-oriented) frameworks to capture the knowledge necessary to know when and how to apply those frameworks. Inspired by the work of Christopher Alexander on cataloging patterns used in civil architecture, software engineers started to document proven solutions to recurring problems.

Initially, patterns focused mainly on object oriented design and reuse; the canonical work in this area is the book by the ‘Gang of Four’ [130]. The aim was to let those design patterns capture expert and design knowledge, necessary to know when and how to reuse design and code. Soon, however, the patterns community extended its horizon beyond object-oriented design. Nowadays, patterns exist in many areas, including patterns for analysis (e.g., [125]), architectural design (e.g., [128, 64]), and the development process (e.g., [82, 81]).

Patterns in software development serve two purposes. Patterns are reusable solutions that can be applied to recurring problems. They also form a vocabulary that provides a common frame of reference, which eases sharing architectural knowledge between developers. Although patterns are usually documented according to certain templates, there is not a standard template used for all patterns. The way in which patterns are codified make them very suitable for human consumption, but less so for automated tools.

2.2.1.2 Dynamism-Centric View

A more formal approach to architectural knowledge can be found in discussions on dynamic software architectures. Systems that exhibit such dynamism can dynamically adapt their architecture during runtime, and for example perform upgrades without the need for manual intervention or shutting down. Such systems must be able to self-reflect and ‘reason over the space of architectural knowledge’ [132], which invariably means that – unlike patterns – this architectural knowledge must be codified for consumption by non-human agents.

As the software itself must understand the architectural knowledge, architecture-based adaptation has to rely on rather formal ways of codification. A 2004 survey by Bradbury et al. [50] reveals that almost all formal specification approaches for dynamic software architectures are based on graph representations of the architectural structure. Purely graph-based approaches use explicit graph representations of components and connectors. In those approaches, architectural reconfiguration is expressed with graph rewriting rules. Other approaches, which use implicit graph representations, rely mainly on process algebra or logic to express dynamic reconfiguration of the architecture.

A particular family of formal languages for representing architectures is formed by so-called ‘architecture description languages’ (ADLs). Although not all ADLs are suitable for use in run-time dynamic systems, all ADLs are based on the same component-connector graph-like representation of architectures (cf. [224]).

2.2.1.3 Requirements-Centric View

The architecture is, ultimately, rooted in requirements. Therefore, architectural knowledge plays a role in enabling traceability in the transition from requirements to architecture. But there is an inverse relation too, namely the fact that ‘stakeholders

are quite often not able to specify innovative requirements in the required detail without having some knowledge about the intended solution' [254]. Hence, in order to specify sufficiently detailed requirements, one needs knowledge about the (possible) solutions, which means that requirements and architecture need to be co-developed. This is a subtle, but important difference: the transition-view is a bit older and denotes the belief that problem is followed by solution, whereas the more recent co-development view emphasizes that both need to be considered concurrently.

The relation between requirements and architecture has been a popular subject of discourse since the early 2000s. Although the related STRAW workshop series is no longer organized, many researchers still focus on bridging the gap between requirements and architecture. A 2006 survey by Galster et al. [129] identified and classified the methodologies in this area, including Jackson's problem frames (later extended to 'architectural frames' [262]), goal-oriented requirements engineering, and the twin peaks model for weaving architecture and requirements [235].

2.2.1.4 Decision-Centric View

For many years, software architecture has mainly been regarded as the high-level structure of components and connectors. Many architectural description frameworks, such as the IEEE-1471 standard [155] therefore have a particular focus on documenting the end result of the architecting process.

Nowadays, the view on architecture seems to be shifting from the end result to the rationale behind that end result. More and more researchers agree that one should consider not only the resulting architecture itself, but also the design decisions and related knowledge that represent the reasoning behind this result. All such architectural knowledge needs to be managed to guide system evolution and prevent knowledge vaporization [48].

The treatment of design decisions as first-class entities enables the consideration of a wide range of concerns and issues, including pure technical issues, but also business, political and social ones. Architects need to balance all these concerns in their decision making. To justify the architectural design to other stakeholders, communication of the architectural design decisions plays a key role. In that sense, the decision-centric view is very much related to the (broader) field of design rationale.

2.2.2 So, What Is Architectural Knowledge?

If there's anything clear from the four views on architectural knowledge, it must be that there is not a single encompassing definition of what this knowledge entails. A 2008 survey of definitions of architectural knowledge revealed that most studies circumstantially define architectural knowledge. Those studies that do give a direct definition are of recent date, and all of them have roots in the decision-centric view [43].

The apparent importance of the decision-centric view in discussing and defining architectural knowledge may be explained when we look at the links between this view and the other views; decisions appear to be the linking pin between the different views.

The relation between patterns and decisions is discussed by Harrison et al. Their conclusion is that the two are complementary concepts, and that ‘[u]sing a pattern in system design is, in fact, selecting one of the alternative solutions and thus making the decisions associated with the pattern in the target systems specific context’ [144]. Ran and Kuusela proposed a hierarchical ordering of design patterns in what they call a ‘design decision tree’ [261].

The relation between design decisions and requirements can be approached from two directions. Bosch conceptually divides an architectural design decision into a ‘solution part’ and a ‘requirements part’ [48]. The requirements part represents the subset of the system’s requirements to which the solution part provides a solution. Van Lamsweerde, on the other hand, argues that for alternative goal refinements and assignments, ‘decisions have to be made which in the end will produce different architectures’ [199].

Formal graph-based architecture representations are especially suitable for automated reasoning. In other words, those representations enable automated agents either take design decisions themselves [50], or to inform human architects about potential problems and pending decisions [267].

Decisions may indeed be an umbrella concept that unify parts of those different views on architectural knowledge, because they closely relate to various manifestations of architectural knowledge concepts. Of course, there are differences between the views too: patterns are individual solution fragments, formal representations focus on the end result only, and requirements engineering is more occupied with problem analysis than solution exploration. If we want to better compare the different manifestations of architectural knowledge, it helps to distinguish between different types of architectural knowledge. Two distinctions are particularly useful: tacit vs. explicit knowledge, and application-generic vs. application-specific architectural knowledge.

Nonaka and Takeuchi draw the distinction between tacit and explicit knowledge [234] (see also Chap. 1). This distinction is applicable to knowledge in general, and its application to architectural knowledge allows us to distinguish between the (tacit) knowledge that an architect and other stakeholders built up from experience and expertise, and the (explicit) architectural knowledge that is produced and codified – for example in artifacts such as architecture descriptions.

The distinction between application-generic and application-specific architectural knowledge has been proposed by Lago and Avgeriou [195]. This distinction, which is not necessarily applicable to other knowledge than ‘architectural’ knowledge, allows us to distinguish between (application-generic) knowledge that is “a form of library knowledge” that “can be applied in several applications independently of the domain” and (application-specific) knowledge that involves “all the decisions that were taken during the architecting process of a particular system and the architectural solutions that implemented the decisions”. In summary,

19.25

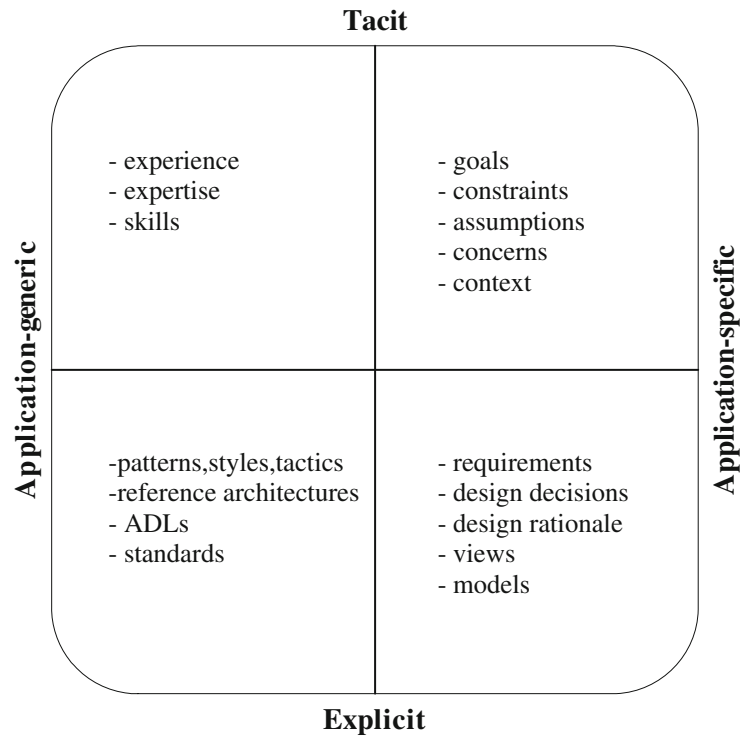


Fig. 2.1 Architectural knowledge categories

application-generic knowledge is all knowledge that is independent of the application domain, application-specific knowledge is all knowledge related to a particular system.

Since the two distinctions are orthogonal, a combination of the two results in four main categories of architectural knowledge, which are depicted in Fig. 2.1. That figure also provides examples of the type of architectural knowledge that fits each of the four categories.

- Application-generic tacit architectural knowledge includes the design knowledge an architect gained from experience, such as architectural concepts, methodologies, and internalized solutions.
- Application-specific tacit architectural knowledge concerns contextual domain knowledge regarding forces on the eventual architectural solution; it includes business goals, stakeholder concerns, and the application context in general.
- Application-generic explicit knowledge is design knowledge that has been made explicit in discussions, books, standards, and other types of communication. It includes reusable solutions such as patterns, styles and tactics, but also architecture description languages, reference architectures, and process models.

- Application-specific explicit architectural knowledge is probably the most tangible type of architectural knowledge. It includes all externalized knowledge of a particular system, such as architectural views and models, architecturally significant requirements, and codified design decisions and their rationale.

2.3 Philosophies of Architecture Knowledge Management

Knowledge from each of the four architectural knowledge categories can be converted to knowledge in another (or even in the same) category. This conversion lies at the basis of different architecture knowledge management philosophies. For some, architecture knowledge management may be mainly intended to support the transition from application-generic to application-specific knowledge. For others, the interplay between tacit and explicit knowledge may be the essence of architecture knowledge management.

Nonaka and Takeuchi defined four modes of conversion between tacit and explicit knowledge: socialization (tacit to tacit), externalisation (tacit to explicit), internalisation (explicit to tacit), and combination (explicit to explicit; cf. Chap. 1). Based on the distinction between application-generic and application-specific knowledge, we can define four additional modes of conversion:

- *Utilization* is the conversion from application-generic to application-specific knowledge. It is a common operation in the architecting process where background knowledge and experience are applied to the problem at hand.
- *Abstraction* is the conversion from application-specific to application-generic knowledge. In this conversion, architectural knowledge is brought to a higher level of abstraction so that it has value beyond the original application domain.
- *Refinement* is the conversion from application-specific to application-specific knowledge. Here, the architectural knowledge for a particular application is analyzed and further refined and related.
- *Maturement* is the conversion from application-generic to application-generic knowledge. It signifies the development of the individual architect as well as the architecture field as a whole; a kind of learning where new generic knowledge is derived and becomes available for application in subsequent design problems.

In total, there are 16 architectural knowledge conversions. Each conversion is formed by pairing one of the four conversions between tacit and explicit knowledge with one of the four conversions between application-generic and application-specific knowledge. Together, the 16 conversions form a descriptive framework with which different architecture knowledge management philosophies can be typified.

We saw earlier that the four views on architectural knowledge can all be related to decision making. At the same time, we saw that there are also obvious differences between those views. Those differences are very much related to the different architecture knowledge management philosophies they are based on.

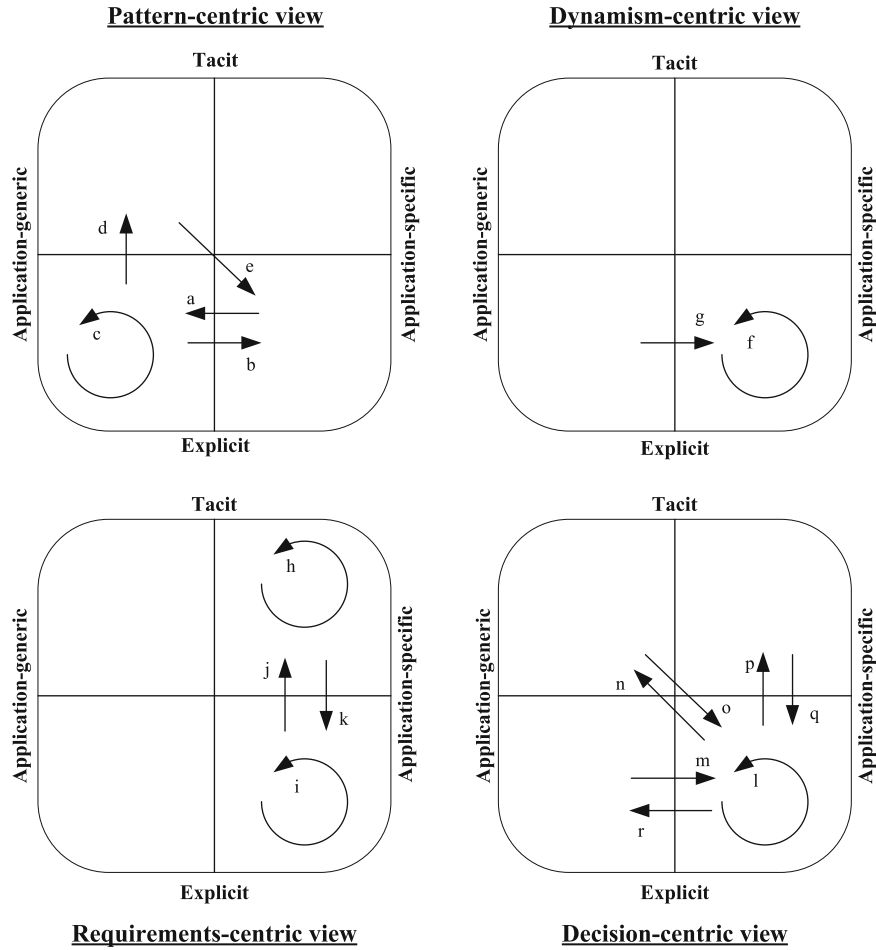


Fig. 2.2 AK management philosophies

The *pattern-centric* view, for example, is mainly geared towards the development of a shared vocabulary of reusable, abstract solutions. As such, the development of a shared tacit mental model is a major goal. This goal is achieved by sharing application-generic patterns that are mined from application-specific solutions. A second goal is the development of libraries of reusable solutions, which is made possible by maturation of documented patterns by cataloging them. The knowledge management philosophy in this view is primarily based on the following architectural knowledge conversions, which are also visualized in Fig. 2.2:

- (a) *Abstraction and combination.* The most obvious example of this conversion is the process of pattern mining. Patterns are inherently abstractions. They are mined from existing architectural solutions and described in a clear and structured way to improve the reusability.

- (b) *Utilization and combination.* One of the ways in which patterns can be reused in new designs is by looking them up in books, catalogs, or other repositories. Architects who wish to design a system comprised of several independent programs that work cooperatively on a common data structure could look up one of the many books available on architectural patterns (e.g., [64]) to find out that the ‘blackboard’ pattern is just what they need. Grady Booch works on the creation of a Handbook of software architecture, of which the primary goal is “to fill this void in software engineering by codifying the architecture of a large collection of interesting software-intensive systems, presenting them in a manner that exposes their essential patterns and that permits comparisons across domains and architectural styles” [47].
- (c) *Maturement and combination.* Documented patterns may be organized in pattern catalogs. These catalogs present a collection of relatively independent solutions to common design problems. As more experience is gained using these patterns, developers and authors will increasingly integrate groups of related patterns to form so-called pattern languages [276]. Although each pattern has its merits in isolation, the strength of a pattern language is that it integrates solutions to particular problems in important technical areas. An example is provided by Schmidt and Buschmann in the context of development of concurrent networked applications [275]. Each design problem in this domain – including issues related to connection management, event handling, and service access – must be resolved coherently and consistently and this is where pattern languages are particularly helpful.
- (d) *Maturement and internalisation.* Experienced architects know numerous patterns by heart. Such architects have no trouble discussing designs in terms of proxies, blackboard architectures, or three-tier CORBA-based client-server architectures. Their exposure to and experience with those patterns has led to internalised and matured pattern knowledge. The consequent shared, tacit vocabulary can be employed at will, without the need to look up individual patterns in order to understand what the other party means.
- (e) *Utilization and externalisation.* When an architect has internalized several patterns, the use of those patterns as a means for communication or design is a combination of utilization (of the pattern) and externalisation (of the knowledge about the pattern and its existence).

While the pattern-centric view aims to support the development of tacitly shared application-generic architectural knowledge, the *dynamism-centric* view is much more focused on explicit application-specific architectural knowledge. Although some application-generic knowledge is utilized, the actual reasoning and reconfiguration uses application-specific knowledge. This architecture knowledge management philosophy is therefore primarily based on two conversions:

- (f) *Refinement and combination.* Corresponds to the formal reasoning over codified architectural solutions in terms of models that are consumable by non-human agents. According to Bradbury et al., in a self-management system all dynamic architectural changes have four steps (initiation of change, selection of

architectural transformation, implementation of reconfiguration, and assessment of architecture after reconfiguration) which should all occur within the automated process [50].

- (g) *Utilization and combination.* Corresponds to formally specifying the application's components and connectors in generic languages such as ADLs or other graph representations. Medvidovic and Taylor propose a comparison and classification framework for ADLs, enabling the identification of key properties of ADLs and to show the strength and weaknesses of these languages [224]. In their comparison, Medvidovic and Taylor point out that at one end of the spectrum some ADLs specifically aim to aid architects in understanding a software system by offering a simple graphical syntax, some semantics and basic analyses of architectural descriptions. At the other end of the spectrum, however, ADLs provide formal syntax and semantics, powerful analysis tools, model checkers, parsers, compilers, code synthesis tools and so on.

For the *requirements-centric* view, the primary goal is tracing knowledge about the problem to knowledge about the solution. In co-development, an additional goal is to connect knowledge about problem and solution so that a stakeholder's image of the problem and solution domain is refined. In this view, the primary architectural knowledge conversions are:

- (h) *Refinement and socialization.* Especially in co-development, architects and stakeholders will interact to align system goals and architectural solutions. Such interaction between 'customer' and 'product developer' is a prime example of a situation in which tacit knowledge is shared (cf. [234]). Pohl and Sikora discuss the need for co-development of requirements and architecture. They argue that such co-design is only possible if there is sufficient knowledge about the (course) solution when defining (detailed) system requirements: "instead of defining requirements based on implicit assumptions about the solution, requirements and architectural artifacts need to be developed concurrently" [235].
- (i) *Refinement and combination.* Adding and maintaining traceability from requirements to architecture is a refinement step that combines explicit knowledge about elements from the problem and the solution space. Several techniques exist to guide the transition from requirements engineering to software architecture. In their 2006 survey, Galster et al. use architectural knowledge as knowledge about (previous) architectural solutions that can be reused when encountering similar requirements in a new project. They present patterns as a useful container for these reusable assets. Another approach for guiding the transition between requirements and architecture is that of feature-solution graphs [56]. De Bruin and Van Vliet use architectural knowledge as knowledge about quality concerns (represented as features) and solution fragments at the architectural level (represented as solutions). These are modeled together as Feature-Solution graphs in order to guide the explicit transition (or alignment) between the problem and solution world.
- (j) *Refinement and externalisation.* Externalisation of application-specific tacit knowledge – such as concerns, goals, and the like – is an important part of

the requirements process. Externalisation of tacit knowledge (problem-related as well as solution-related) is a precondition for maintaining traceability.

- (k) *Refinement and internalisation.* The interaction between architects and stakeholders is not purely a matter of socialization. In co-development, for example, specifications and design artifacts play a major role as well and may be an aid to let the parties ‘re-experience’ each other’s experiences (cf. [234]). This internalization of explicit application-specific architectural knowledge may consequently lead to ‘new ideas and insights concerning both the envisioned system usage and the architectural solution’ [254].

Finally, the essential philosophy of the *decision-centric* view is externalizing the rationale behind architectural solutions (i.e. ‘the why of the architecture’) that can then be internalized and shape other people’s mental models, so as to prevent knowledge vaporization. The architectural knowledge conversions central to this philosophy are:

- (l) *Refinement and combination.* Corresponds to reasoning about codified architectural solutions, which need not be fully automated but may involve decision support. One such decision support approach is introduced by Robbins et al., who introduce the concept of ‘critics’ [267]. Critics are active agents that support decision-making by continuously and pessimistically analyzing partial architectures. Each critic checks for the presence of certain conditions in the partial architecture. Critics deliver knowledge to the architect about the implications of, or alternatives to, a design decision. Often, critics simply advise the architect of potential errors or areas needing improvement in the architecture. One could therefore see critics as an automated form of the backlog that architects use in the architecting process to acquire and maintain overview of the problem and solution space [146]. Another approach that fits this conversion is the Archium tool proposed by Jansen et al. which is aimed at establishing and maintaining traceability between design decision models and the software architecture design [163].
- (m) *Utilization and combination.* This conversion amounts to the reuse of codified, generic knowledge (decision templates, architectural guidelines, patterns, etc.) ‘to take decisions for a single application and thus construct application-specific knowledge’ [195] (see Chap. 12).
- (n) *Internalisation and abstraction.* Based on experience and expertise, an architect may quickly jump to a ‘good’ solution for a particular problem. Such a good solution can be a combination of several finer grained design decisions. This combination of design decisions may become so common for the architect that the solution is no longer seen as consisting of individual decisions. It may be hard for the architect to reconstruct why a certain solution fits a particular problem; the architect ‘just knows’.
- (o) *Utilization and externalisation.* When a solution has been internalized, and the architect ‘just knows’ when to apply it, it becomes difficult to see which other solutions are possible. Part of the decision-centric philosophy (e.g., in [56]) is therefore to reconstruct and document the constituting design decisions.

- (p) *Refinement and internalisation.* This ‘consumption’ of architectural knowledge takes place when people want to ‘learn from it or carry out some quality assessment’ [195].
- (q) *Refinement and externalisation.* The rationalization of taken architectural decisions, i.e. reconstruction and explanation of the ‘why’ behind them, is a crucial part of the decision-centric philosophy in which tacit knowledge is made explicit. In this respect, the software architecting can apply the best practices known from the ‘older’ and well-known field of design rationale. Regli et al. present a survey of design rationale systems, in which they distinguish between process-oriented and feature-oriented approaches [264]. Process-oriented design rationale systems emphasize the design rationale as a ‘history’ of the design process, which is descriptive and graph-based. A well-known example is the Issue-Based Information System (IBIS) framework for argumentation. A feature-oriented approach starts from the design space of an artifact, where the rules and knowledge in the specific domain must be considered in design decision making. Often these type of design rationale systems offer support for automated reasoning. A more recent survey on architecture design rationale by Tang et al. provides information about how practitioners think about, reason with, document and use design rationale [314]. It turns out that although practitioners recognize the importance of codifying design rationale, a lack of appropriate standards and tools to assist them in this process acts as barrier to documenting design rationale. Fortunately, the field of design rationale is working hard on developing more mature support. Recent developments have led to the creation of mature tooling such as Compendium and SEURAT [63], and models such as AREL [316] (see also Chap. 9).
- (r) *Abstraction and combination.* The construction of high-level structures (templates, ontologies, models) to capture and store architecture design decisions. Various approaches to codify architectural design decisions in such a way have been reported. Tyree and Akerman present a template to codify architectural design decisions together with their rationale and several other properties relevant to that decision, such as the associated constraints, a timestamp, and a short description [325]. Kruchten proposes a more formal approach of codifying design decisions by using an ontology [190] (see Chap. 3). Ran and Kuusela present work on design decision trees [261].

2.4 State-of-the-Art in Architecture Knowledge Management

Based on the discussion of the four philosophies on architecture knowledge management and the primary architectural knowledge conversions of these philosophies, we can identify a few single-philosophy approaches to architecture knowledge management:

- *Rationale management systems.* In order to achieve refinement and externalisation of architectural knowledge, design rationale needs to be managed. Various tools and methods have been proposed over the last decade for doing exactly this.

- *Pattern languages and catalogs.* To allow for utilization and combination of architectural knowledge architectural knowledge needs to be categorized in pattern languages or catalogs. From these sources it can be quickly used in specific applications. In addition it enables learning and reasoning.
- *Architecture description languages.* ADLs enable utilization and combination of architectural knowledge in a formal way. Various types of ADLs exist, which range in level of formality and purpose, the latter ranging from aiding understanding of software systems to enabling powerful analyses.
- *Design decision codification.* To support refinement and combination of architectural knowledge much effort has been put in methods, tools, and techniques to codify architectural knowledge concepts. Design decisions are central to these methods and tools, but also related concepts such as rationale and alternative solutions are captured.

We saw earlier how the concept of ‘decisions’ seem to be an umbrella concept for all views on architectural knowledge. We can now explain this better: ‘mature’ architecture knowledge management unifies conversions from all architecture knowledge management philosophies, and is not merely limited to just one philosophy. This concept, which we call ‘*decision-in-the-large*’, is related more to the architecting process than to pure rationale management (which we could call ‘*decision-in-the-narrow*’), even though codifying rationale and preventing knowledge vaporization has been one of the prime drivers for the decision-centric philosophy. A focus on ‘decision-in-the-large’ seems driven more by architectural knowledge use cases than by anything else, often under the concept ‘knowledge sharing’. A classification of such use cases is presented by Lago et al. [195], which distinguishes between architecting, sharing, assessing, and learning.

Obviously, some ‘decision-in-the-large’ approaches evolved from ‘decision-in-the-narrow’ approaches. But the former starts to play a more prominent role in the other philosophies, and hence the other views, too. In recent years, a shift towards more overarching state-of-the-art approaches can be observed. Four main trends for architecture knowledge management can be identified that mostly focus on specific use cases for architecture knowledge management. These trends will be elaborated upon in Sections. 2.4.1–2.4.4.

2.4.1 *Sharing Architectural Knowledge*

The trend of sharing architectural knowledge is focused on providing support for management of various types of architectural knowledge (both application-generic and application-specific). De Boer et al. propose a core model of architectural knowledge that provides further insight in what architecting entails (e.g. how design decisions are made) and which knowledge concepts are worth focusing on in support for architecture knowledge management [44]. This includes concepts that have their origin in the pattern-centric, requirements-centric and decision-centric view.

From knowledge management literature it is known that knowledge sharing can be achieved through codification or personalization [143]. Farenhorst et al. have come to the conclusion that a hybrid strategy might work best. They propose a platform for architectural knowledge sharing that combines codification techniques (design decision repositories, document management facilities) with personalization mechanisms (yellow pages, discussion forums) in order to enable Just-In-Time architectural knowledge: delivery of and access to the right architectural knowledge, to the right people, at the right time [113]; see also Chap. 8.

Another platform that allows managing different types of architectural knowledge concepts is the process-based architecture knowledge management environment (PAKME) proposed by Ali Babar et al. [8]. This environment allows storing application-generic architectural knowledge (such as general scenarios, patterns, and quality attributes), and application-specific architecture knowledge (such as concrete scenarios, contextualized patterns, and quality factors).

2.4.2 Aligning Architecting with Requirements Engineering

We observe a trend in architecture knowledge management literature towards aligning the architecting process with requirements engineering, since the two practices seem to have much in common. Pohl et al. propose COSMOD-RE, a method for co-designing requirements and architecture [253]. This method supports the development of detailed requirements based on the specified architecture and the specified goals and scenarios. This co-design focus is fundamentally different from the focus on traceability from requirements to architecture, a predominant focus in the requirements-centric view.

With the transition from traceability to co-development/co-design the requirements engineering community appears to look beyond their own philosophy. In an attempt to better understand the relationship of requirements and architecture, De Boer and Van Vliet explore similarities between the two [46]. Based on their study they argue that there is no fundamental difference between architecturally significant requirements and architectural decisions, which also pleads for integrated methods and tools for architecture knowledge management that overarch requirements engineering and architecting.

2.4.3 Intelligent Support for Architecting

This trend focuses on specific architecting use cases related to intelligent and/or real-time support for architects. One of the state-of-the-art approaches focuses on providing architects or reviewers with a reading guide when looking for specific architectural knowledge [45]. To save time, architects use this intelligent discovery

method to quickly get to the important knowledge while skipping less relevant documentation.

Other intelligent support approaches focus on modeling architectural knowledge concepts in such a way that learning and reasoning is stimulated. One example is provided by Kruchten, who proposes an ontology of architectural design decisions [190]. The templates he proposes allow acquiring insights in not only important properties of design decisions (e.g. the status), but also in the relationships between design decisions (e.g. ‘enables’ or ‘conflicts with’). When proper visualization techniques are used this information is very useful for architects in the decision making process, for example to model the backlog [146].

2.4.4 Towards a Body of Architectural Knowledge

The last trend relates to establishing a body of architectural knowledge, to enable both learning and reuse. Recently, several approaches for setting up such a body of knowledge have been introduced. Lenin Babu et al. propose ArchVoc, an ontology for software architecture [24]. Based on knowledge from major textbooks on software architecture and by parsing parts of the web, they have constructed a software architecture vocabulary for reuse purposes. Their ontology of software architecture enables the architect in understanding the existing best practices and the relationships between them and also provide a means to apply them to the new systems to be developed.

The need for cataloging architectural knowledge is also expressed by Shaw and Clements, who argue that Booch’ Handbook (cf. Sect. 2.3) “can provide important exemplars, but engineers also need reference material that organizes what we know about architecture into an accessible, operational body of knowledge” [295]. Chapter 12 gives a successful example hereof in the area of SOA infrastructure. This body of knowledge is thus not comprised of only patterns, but also other types of explicit application-generic architectural knowledge that can be utilized effectively. In an attempt to further define these types of architectural knowledge, Clements et al. have conducted empirical research to find out what set of duties, skills and knowledge is most important for an architect [73]. Codification of this architectural knowledge is perceived as “the beginnings of a road map for training and mentoring.”

2.5 Justification

Our state-of-the-art overview on architecture knowledge management is the result of an extensive literature review conducted based on a predefined protocol. More details on this protocol can be found in [43]. The main phases executed are discussed in the remainder of this section.

In a systematic literature review of studies that define or discuss ‘architectural knowledge’, we identified 115 such studies [43]. These 115 studies form the basis for our discussion in this chapter. Consequently we will refer to these studies as the set of core studies (C).

Since we are interested in the community structures that underly the topic of architectural knowledge, we enriched our dataset with bibliographical links. We assume the community structure can be found, or approximated, by taking into account the bibliographical references that various authors make to each others work. Strong communities will display many intra-community references, and relatively few references to work outside the community. There are two types of bibliographical references: references *from* studies in C to other studies, and references *to* studies in C from other studies. We will use B (for ‘bibliography’) to denote studies that are referred to *from* studies in C , and R (for ‘referring’) to denote studies that refer *to* studies in C . Hence, the mapping $R \rightarrow C \rightarrow B$ summarizes the enriched data set used throughout this chapter.¹

To determine B , we simply took all references listed in the 115 studies from C . To determine R , however, we had to do a ‘reverse search’ on the studies in C , since the information needed is not present in C itself. For construction of R , we used the Google Scholar search engine which provides such a reverse search facility through its ‘cited by’ feature. While constructing R and B , we also obtained results not necessarily found in sources from the sources list identified in our systematic review (cf. [43]). We do not see this as a limitation of our methodology. The goal of the dataset enrichment is different from the initial identification of primary studies in the systematic review; in the enrichment we are interested in community structures, and the different communities need not be limited to studies published through and indexed by the sources used for the review.

Together, B , C , and R comprise a ‘social network’ of scholarly publications and their interconnections. We analyzed this network using the Girvan–Newman algorithm [134]. The Girvan–Newman algorithm discovers communities in a graph, based on the ‘betweenness’ of edges, i.e., the number of shortest paths that run through an edge. The algorithm iteratively calculates the edge betweenness and removes the edge with the highest betweenness value, thereby eliminating edges that act as connections between different communities. Eventually, the algorithm results in a fully disconnected graph of ‘communities’ that consist of a single node.

Obviously, a disconnected graph is not the strongest community structure possible. Newman defines the modularity (Q) as a measure of strength of the community structure discovered in a network [233]. High values of Q ($0 \leq Q \leq 1$) indicate networks with a strong community structure. Newman’s empirical evidence shows that local maxima of Q correspond to reasonable community divisions, hence it is good practice to stop the Girvan–Newman algorithm when a (local or global) maximum Q -value has been obtained. In our case, the first local maximum of Q occurred when the graph had been split up in 52 communities, while the global maximum occurred

¹ Note that B , C , and R are not completely disjoint; there are several occurrences of studies from C referring to other studies from C . Also, publications that are referred to from one study in C may themselves refer to other (earlier) studies from C .

at 59 communities ($Q \approx 0.7832$), which is extremely high (according to Newman, values above 0.7 are rare; typical values fall in the range 0.3 – 0.7). Because of the data enrichment process we followed and the way the Girvan–Newman algorithm works, each of these 59 communities consists of at least one study from *C*, plus zero or more publications from either *B* or *R*.

In order to assign meaning to the 59 communities that came out of the algorithm we examined the set of papers for each of these communities in turn, and gave them a label that corresponded best to the papers in that community. Often, the non-core papers in the community did help in further characterizing the community and helped in phrasing a suitable label. When this was more difficult (for example when the non-core papers varied too much in subject), we looked more specifically at the core papers in that community since these actually talk about architectural knowledge and can therefore lead to more fitting for the community name. In the end we ended up with 59 labels for the communities, although some of those did overlap to a certain extent with each other.

In order to find out how exactly the communities had been discovered by the Girvan–Newman algorithm we examined the hierarchical structure of the identified communities. The hierarchical relations capture the order in which the communities have been identified. Based on the order of community-split-ups we could assign names to larger-order (i.e. parent) communities as well.

According to our definition a community should consist of at least two core papers (that talk about architectural knowledge) written by different authors. Based on this rule we further analyzed the data. We limited the number of main communities by removing the single-core-paper ones and the ones consisting of merely papers by the same author(s). In the end this refinement culminated into the four main communities discussed throughout this chapter: pattern-centric, dynamism-centric, requirements-centric, and decision-centric.

2.6 Summary

In this chapter we have analyzed the state-of-the-art in architecture knowledge management, and have shown that the concept of ‘architectural knowledge’ is becoming more prominent in literature. In Sect. 2.2.1 we have identified four main views on architectural knowledge: a pattern-centric view, a dynamism-centric view, a requirements-centric view, and a decision-centric view. The concept of ‘decision’ was found to be an umbrella concept that unifies parts of these views.

To better understand the concept of architectural knowledge in Sect. 2.2.2 we have defined four main categories (or types) of architectural knowledge based on a distinction between tacit and explicit knowledge on the one hand, and application-generic and application-specific architectural knowledge on the other hand. In total 16 knowledge conversions are possible between these four types of architectural knowledge. To articulate the commonalities and differences between the four views, in Sect. 2.3 we stated their main philosophies of architecture knowledge

management and elaborated upon the architectural knowledge conversions that are central for these views. The conversions were further illustrated by examples of related work.

Based on the discussion of the main philosophies we identified four single-philosophy approaches for architecture knowledge management: rationale management systems, pattern languages and catalogs, architectural description languages, and design decision codification. All these approaches have both their origin in and scope on only one of the main philosophies. In recent years, however, a shift towards more overarching approaches can be observed. Four main trends for architecture knowledge management can be identified that mostly focus on specific use cases for architecture knowledge management (e.g. sharing, learning, traceability). This state of the art in architecture knowledge management indicates a shift from ‘decision-in-the-narrow’ to ‘decision-in-the-large’.

We expect the interest in architectural knowledge to keep increasing over the coming years. Although it is unlikely that we will see a unified view on architectural knowledge anytime soon, the observed trends in architecture knowledge management indicate that future developments on managing architectural knowledge may further align the different views. We expect the trend towards ‘decision-in-the-large’ to continue, since both researchers and practitioners aim for a better understanding of what architects do, what their knowledge needs are, and how this architectural knowledge can best be managed in the architecting process.

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

Software Architecture Knowledge Management
Theory and Practice

Ali Babar, M.; Dingsøyr, T.; Lago, P.; van der Vliet, H.
(Eds.)

2009, XX, 279 p., Hardcover

ISBN: 978-3-642-02373-6