The Decision View's Role in Software Architecture Practice

Philippe Kruchten, *University of British Columbia*
Rafael Capilla, *Universidad Rey Juan Carlos*
Juan Carlos Dueñas, *Universidad Politécnica de Madrid*

This is a journey of discovery from software architecture representation to architectural methods, to design decisions, to a decision view, which enables architects to capture architectural design decisions and design rationale as first-class entities.

Software development has to deal with many challenges—increasing system complexity, requests for better quality, the burden of maintenance operations, distributed production, and high staff turnover, to name just a few. Increasingly, software companies that strive to reduce their products' maintenance costs demand flexible, easy-to-maintain designs. Software architecture constitutes the cornerstone of software design, key for facing these challenges. Several years after the "software crisis" began in the mid-1970s,[1] software architecture practice emerged as a mature (although still growing) discipline, capable of addressing the increasing complexity of new software systems. The term software architecture was first coined at a 1969 NATO conference on software engineering techniques, but it wasn't until the late 1980s that software architectures were used in the sense of system architecture.[2]

Today, modern software architecture practices still rely on the principles that Dewayne E. Perry and Alexander L. Wolf enunciated in their lovely, yet simple formula "Architecture = {Elements, Form, Rationale}."[3] Elements are the main constituents of any architectural description in terms of components and connectors, whereas the nonfunctional properties guide the architecture's final shape. Different shapes with the same or similar functionality are possible; they constitute valid design choices by which software architects make their design decisions. These decisions are precisely the soul of architectures. However, they're often neglected during architecting because they usually reside in the architect's mind as tacit knowledge, which is seldom captured and documented in a usable form. Furthermore, as the Rational Unified Process (RUP) states, software architecture practice

> *encompasses significant decisions about*
>
> - *the organization of a software system,*
> - *the selection of the structural elements and their interfaces by which a system is composed with its behavior as specified by the collaboration among those elements, and*
> - *the composition of these elements into progressively larger subsystems.*[4]

For years, architecture practice and research efforts have focused solely on architecture representation itself. For a long time, these practices have exclusively aimed at representing and documenting a system's architecture from different perspectives—the so-called *architectural views*. These views represent different stakeholders' interests as a set of coherent, logical, harmonized descriptions; they're also used to communicate the architecture. *IEEE Standard 1471-2000 Recommended Practice for*

0740-7459/09/$25.00 © 2009 IEEE

Architectural Description of Software-Intensive Systems provides a guide for describing the architecture of complex, software-intensive systems in terms of views and viewpoints.[5] However, it doesn't offer a detailed description of the rationale that guides the architecting process.

This article describes the historic evolution of software architecture representation and the role it can play. We use a set of epiphanies that can guide you from the initial architecture views to a new *decision view*, expressing the need for capturing and using architectural design decisions and design rationale as first-class entities. When we explicitly record and document design decisions, new activities arise during the architecting process; this architectural knowledge (AK) constitutes a new crosscutting view that overlaps the information described by other views.

First Epiphany: Architectural Representation

Before 1995—that is, prior to the notion of the architecture view—software designers did architecting, but the demand for large complex systems brought new design challenges. Such systems' intrinsic complexity, with different structures entangled in different levels of abstractions, was organized into a set of architecture views that tried to describe the system from different perspectives, according to different users' needs.

As a result, Philippe Kruchten proposed architecture views in his "4+1" view model to provide a blueprint of the system from different angles.[6] That model uses four views to describe the design concerns of different stakeholders, plus a use-case view (the +1) that overlaps the others and relates the design to its context and business goals (see Figure 1). Many Rational Software consultants used the set of views in the 4+1 view model in large industrial projects as part of the RUP approach. Similarly, Siemens developed the Siemens Four-Views (S4V) method, based on best architectural practices for industrial systems.[7] The S4V method aimed to separate engineering concerns to reduce the complexity of the design task.[8]

In 1995, we proposed views that helped architects identify all the influencing factors they can use to identify the key architectural challenges and to develop design strategies for solving the issues by applying one or more views. In such contexts, we evaluate design decisions (that is, strategies applied to particular views) according to constraints or dependencies on other decisions. The Software Engineering Institute proposed a classification based on views and view types that highlights the importance of documenting design decisions. However, it gave no details on how to do this and failed to define adequate processes for capturing and documenting those decisions.[9] Nick Rozanski and Eoin Woods defined up to six viewpoints that clarify the most important architectural aspects or elements of information systems that are relevant for stakeholders.[10] In the mid-1990s, architecture research focused on design description and modeling, with little agreement on notations for architecture representation.

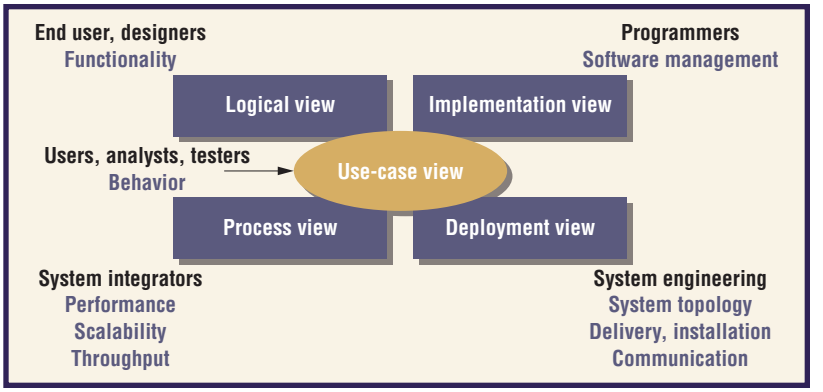

Figure 1. The "4+1" architecture view model.[6] Four views describe the design concerns of different stakeholders. A use-case view overlaps the others and relates the design to its context and its business goals.

Second Epiphany: Architectural Design

The period from 1996 to 2006 brought complementary techniques in the form of architectural methods, many of them derived from well-established industry practices. Methods such as IBM's RUP, Philips' BAPO/CAFCR (Business-Architecture-Process-Organization method and its Customer, Application, Functional, Conceptual, and Realization views), Siemens' S4V, Nokia's ASC (Architectural Separation of Concerns), and the Software Engineering Institute's ATAM (Architecture Trade-off Analysis Method), SAAM (Software Architecture Analysis Method), and ADD (Attribute-Driven Design) are now mature practices for analyzing, synthesizing, and evaluating modern software architectures. In some cases, they're backed by architectural description languages, assessment methods, and stakeholder-focused decision-making procedures. Because many of the design methods were developed independently,[8] they exhibit certain similarities and differences motivated by the specific nature, purpose, application domain, or organization size for which they were developed. In essence, they cover the key phases of the architecting activity but are performed in different ways.

Common to some of these methods is the use of design decisions that are evaluated during the architecture's construction. Groups of stakeholders, under architects' guidance, elicit these decisions, but

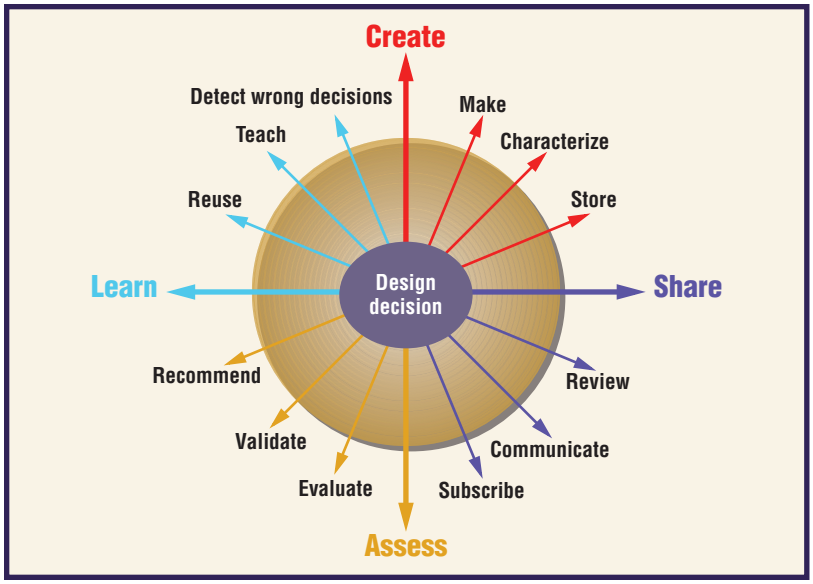

Figure 2. The four main activities—Create, Share, Assess, and Learn—and the subactivities involved in the creation and use of design decisions and design rationale. Each of the four colors shown indicates a main category (for example, "Create") and its related, smaller subactivities (in this case, "Make," "Characterize," and "Store").

the ultimate decision makers are the architects—often a single person or a small group. Unfortunately, design decisions and their rationale still aren't considered first-class entities because they lack an explicit representation. As a result, software architects can't revisit or communicate the decisions made, so in most cases the decisions vanish forever.

Reasons for Design Rationale

In 2002, Ioana Rus and Mikael Lindvall wrote, "The major problem with intellectual capital is that it has legs and walks home every day."[11] Software organizations suffer the loss of this intellectual capital when their experts leave. The same happens in software architecture when the reasoning required for understanding a particular system is unavailable and hasn't been explicitly documented. In 2004, Jan Bosch stated that "we do not view a software architecture as a set of components and connectors, but rather as the composition of a set of architectural design decisions."[12] The lack of first-class representation of design rationale in current architecture view models led to the need to include decisions as first-class citizens that should be embodied within the traditional architecture documentation.

There are several benefits of using design rationales in architecture to explain why a particular design choice was made or to know which design alternatives have been evaluated before making the final design choice. One medium- to long-term benefit is avoiding architecture-recovering processes, which are used mostly to retrieve decisions when an architecture's design, documentation, or even creators are no longer available. Maintaining and managing this AK requires continuous attention to keep the changes in the code and the design aligned with the decisions, and to use these to bridge the software architecture gap.

In this new context, Perry and Wolf's old ideas[3] become relevant for upgrading the software architecture concept by explicitly adding the design decisions that motivate the creation of software designs. Together with design patterns and assumptions, design decisions are a subset of the overall AK that's produced during architecture development. Most of the tacit knowledge hidden in the architects' minds should be made explicit and transferable into a useful form, easing the execution of distributed and collective decision-making processes. The formula Architecture Knowledge = Design Decisions + Design, recently proposed by Kruchten and his colleagues,[13] modernizes Perry and Wolf's formula and considers design decisions part of the architecture.

Third Epiphany: Architectural Design Decisions

Architecture decisions are seldom rigorously documented. Explicitly documenting key design decisions is pretty rare and typically justified only on political and economic grounds or even sometimes fear. So, our third epiphany highlights the need to deal with the representation, capture, management, and documentation of the design decisions made during architecting.

Active research from 2004 to 2008 has produced a significant number of approaches for representing and capturing architectural design decisions, and has defined new roles and activities for supporting the creation and use of this AK. Several approaches use template lists of attributes to describe and represent design decisions as first-class entities.[13–15] One approach emphasizes categorizing different types of dependencies between decisions as valuable, complementary information for capturing useful traces—information that developers can use, for instance, during maintenance to estimate the impact when a decision is added, removed, or changed.[13] Another approach advocates using flexible approaches that employ mandatory and optional attributes for knowledge capture that can be tailored to specific organizations.[15] Others have proposed ontologies to formalize tacit knowledge and make visible the relationships between the decisions and other artifacts of the software life cycle.[13] The field of product-family engineering, or product lines, has yielded a large amount of work about specification, modeling, and automation of design decisions applied to describing and selecting a product line's common and specific elements.[16] For product lines,

knowledge is codified in an operational manner as derivation processes are automated.

New Architecting Activities

Several authors have recently contributed models, methods, and tools that encourage design decisions in both software architecture and software engineering.[17] Because architecture modeling isn't isolated from decision making, new processes must be carried out in parallel with typical modeling tasks. Hence, architecting is highly impacted by these new activities that deal with the creation and use of design decisions.

So, as decision makers, software architects must assume new roles as knowledge producers and consumers in a social process and must perform a variety of new activities. Figure 2 (inspired by a technical report by Patricia Lago and Paris Avgeriou of the first Shark [Sharing and Reusing Architectural Knowledge] workshop[18]) illustrates these two aspects to articulate the decisions made and the architecture resulting from these decisions. For instance, architects capture decisions ("Create") that lead to a particular architecture. In this phase, architects make decisions, characterize them in usable form, and link them to design artifacts. Once the architecting team has created a first version of the architecture, they can share the design with other stakeholders and, for instance, review the status of the architecture. During maintenance, the current architecting team might need to evaluate past decisions and recommend whether they were right or wrong. Because the architecture is continuously evaluated, assessment procedures can occur at different stages of architecture development (when decisions are first made or after). Also, less expert architects can learn from decisions made by others; if they detect wrong decisions, they must fix or replace them with new decisions and modify the architecture accordingly. As a result, a perfect alignment between decisions and design can be achieved.

Additional subactivities refine the main ones shown in Figure 2, but our aim here is just to explain that parallel, complementary activities related to the reasoning process directly influence the architecture-modeling tasks. We justify the separation between knowledge producers and knowledge consumers on the basis of the distinction between architecting for the first time and maintaining the architecture over time.

Impact and Use

Our third epiphany has a strong impact on current architecting practices: two empirical studies have already reported on the value of capturing and using design decisions, and they provide some specific results:

- Design decisions and rationales, considered different types of knowledge for representing and recording design information, might not have the same value or importance for all stakeholders.[19] So, we should decide which type of knowledge would better fit each type of user.
- The effort of capturing decisions during the early development stages really pays off only in later maintenance and evolution phases, so no great return on investment should be expected when decisions are captured for the first time.[20] The experiences described in this report also highlight the benefits of using specific tool support for capturing, managing, and documenting architectural design decisions.

Another visible impact on practice is related to the documentation by means of the traditional views as described in the standard *IEEE Std. 1471-2000*.[5] Its successor, known as ISO/IEC 42010 and currently under review, expands it with AK concepts, including concern, design decision, and rationale.

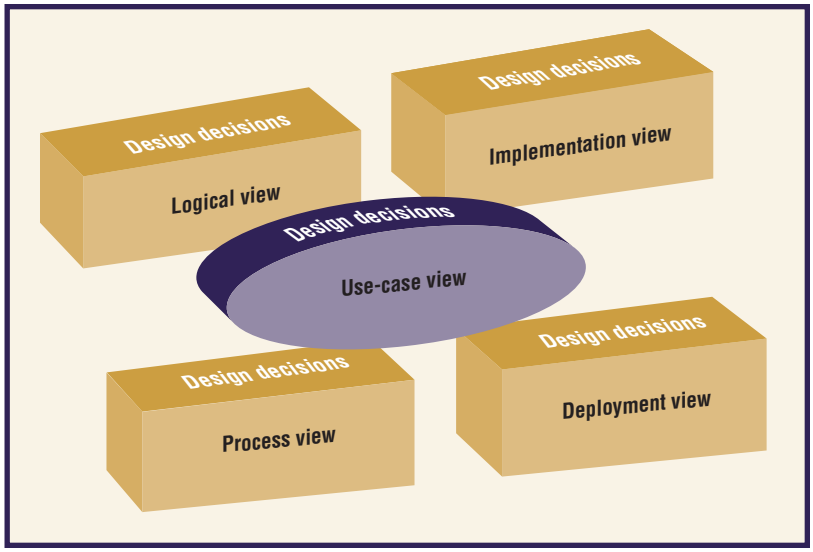

Figure 3. The "decision view" embedded in the 4+1 view model. This new perspective superimposes the design rationale that underlies and motivates the selection of concrete design options.

The Texture of a Decision View

A complementary perspective in which decisions are entangled with design for each architectural view has led us to think about a decision view.[21] This new perspective extends the traditional views by superimposing the design rationale that underlies and motivates the selection of concrete design options. Figure 3 depicts a graphical sketch of the

Long-term benefits and reduced maintenance costs should motivate users to capture the design rationale.

decision view, which incorporates design decisions in the 4+1 view model.

The traditional representation of architectures in terms of views and viewpoints varies when decisions have to be described. Architects interested in capturing decisions and rationale should know how to build a decision view—that is, how to understand and represent the texture of decisions. As a first approach, we can refer to the classic architectural assessment methods, which mostly rely on the development of scenarios, their projection against several candidate architectures, and the addition of information to the architectural components. The architect then aggregates this information and evaluates it for each candidate architecture.

Another possible approach is based on a study of architectural assessment and definition of design decisions on a product-line architecture for medical equipment, in which the decisions related to the economic impact of changing each architectural component.[20] The authors focused on each component's economic attributes in the implementation view (from the 4+1 model), and their decision view consisted of the decisions, rationale, and actual data on the architectural components.

Focusing on the capture and representation of decisions, as a guide to help architects document the decisions in their architectures, we propose these steps:

1. Decide which information items are needed for each design decision (such as the decision's name, description, rationale, pros and cons, status, and category). Then, decide which representation system will better handle the recording and organization of the decisions (that is, as templates or ontologies). Select a strategy (such as codification, personalization, or a hybrid strategy) to capture the items.
2. For each decision, define links to the requirements that motivate it.
3. If you must evaluate alternative decisions, provide mechanisms to change the decision's status (such as approved, rejected, or obsolete) and category (such as alternative or main).
4. If a decision depends on previous ones, define these relationships to support internal traceability among them.
5. Once you've made a set of significant decisions, link them to the architecture that results from such decisions. These links provide the connection to traditional architecture views.
6. After making and capturing all the decisions, share them through communication and documentation mechanisms.

We could add extra items and functionality to this list (for example, supporting the evolution of decisions), but we believe we've listed enough to help you quickly start capturing design decisions and their underpinning rationale alongside their architectures.

Challenges and Benefits

The explicit capture and documentation of design decisions will bring new challenges, but in most cases we see these as benefits derived from using architecture development decisions. Here's a short list of the expected challenges and benefits:

- Decisions enhance traceability between software engineering artifacts produced across the software life cycle. Forward and backward traces facilitate our understanding of the root causes of changes and help us better estimate change impact analysis.
- Capturing the dependencies between decisions supports impact analysis when we add, modify, or remove a decision.
- Documented decisions facilitate our general understanding of a system, which is particularly useful during staff turnover.
- Documented decisions facilitate knowledge sharing and assessment processes because users can easily review the rationale of past decisions.
- Learning activities can use previous knowledge for assessing novice software architects in their professional careers.
- Leveraging tacit AK into formal documentation requires understanding and performing many of the activities described in Figure 2.

The adoption barrier for capturing design rationale can be high because of the intrusiveness of these new activities, listed in Figure 2. So, the overhead required during the creation of these decisions should pay off during maintenance, because knowledge of key design decisions avoids the need to reverse architecture descriptions from code, particularly in staff turnover situations or rapid software evolution. Long-term benefits and reduced maintenance costs should motivate users to capture the design rationale, particularly in successive iterations of the system as it evolves.[21] Hence, the broad impact of capturing and using architecturally significant design decisions affects not only a design's evolution but also the evolution and maintenance of the decisions base itself. This issue often emerges during reviews, where major changes affect the design. Like other key

Tools Supporting Design Rationale

As Allen H. Dutoit and his colleagues pointed out in *Rationale Management in Software Engineering*,[1] the design rationale movement began in the early 1970s with Horst Rittel's Issue-Based Information System (IBIS), which supported design rationale in general. The IBIS approach and its successor gIBIS were applied to large-scale projects in the '70s and '80s. IBIS-based approaches included some basic features supporting the design rationale and discussions on the recording of controversial questions that arise in design. On the basis of Rittel's approach, other tools such as PHI (Procedural Hierarchy of Issues), QOC (Questions, Options, and Criteria), and DRL (Design Representation Language) appeared in the field as extensions of the IBIS tool. Other tools (Scram, C-ReCS, Seurat [www.users.muohio.edu/burgeje/SEURAT], Sysiphus [http://sysiphus.informatik.tu-muenchen.de], and Drimer) developed between 1992 and 2004, provide simple solutions to manipulate knowledge and record decisions for a broad number of software engineering processes.[1] Since 2005, active research has produced a number of tools supporting design rationale in software architecture.

Here, we identify five representative research prototype tools for capturing, using, managing, and documenting architectural design decisions.

Archium (www.archium.net) is a Java extension that provides traceability among a wide range of concepts (such as requirements, decisions, architecture descriptions, and implementation artifacts) that are maintained during the system life cycle. The Archium tool suite contains a compiler, a runtime platform, and a visualization tool. The compiler turns Archium source files into executable models for the runtime platform. The visualization tool uses the runtime platform to visualize and make accessible the architectural knowledge (AK).

The Architecture Rationale and Element Linkage (AREL, www.ict.swin.edu.au/personal/atang/AREL-Tool.zip) is a UML-based tool to help architects create and document architectural designs with a focus on architectural decisions and design rationale. AREL captures three types of AK: design concerns, design decisions, and design outcomes. These knowledge entities are represented as standard UML entities and linked to show their relationships.

The Process-Based Architecture Knowledge Management Environment (PAKME, http://193.1.97.13:8080) is a Web-based tool that supports collaborative knowledge management for the software architecture process. It's built on top of the Hipergate open source groupware platform. PAKME's features can be categorized into four AK management services: acquisition, maintenance, retrieval, and presentation.

The Architecture Design Decision Support System (ADDSS, http://triana.escet.urjc.es/ADDSS) is an ongoing Web-based research prototype that captures design decisions using a template list of mandatory and optional attributes. This tool supports a combined strategy of codification and personalization. Decisions are related to requirements and architectures. The tool provides an automatic reporting system that produces documents containing the decisions made for a given architecture, the trace relationships from decisions to requirements and architectures, and the trace relationships between decisions. In addition, ADDSS users can navigate and visualize the architectures and decisions, showing the system's evolution over time.

The Knowledge Architect (http://search.cs.rug.nl/griffin) is a tool suite for capturing, managing, and sharing AK using a server and an AK repository. It's accessed by three plug-in clients: a Word client to capture and manage AK in MS Word documents, a client that captures and manages the AK of quantitative architectural analysis models using MS Excel, and a visualization tool called the Knowledge Architect Explorer that supports the analysis of the captured AK. This tool enables the exploration of the AK by searching and navigating through the web of traceability links among the knowledge entities.

Reference

1. A.H. Dutoit et al., eds., *Rationale Management in Software Engineering*, Springer, 2006.

activities, recording the history of decisions is another challenge requiring in-depth treatment.

The software architecture community's perception that architectural design decisions are intangible and difficult to capture and communicate is changing as a result of recent research. That research is leading to a new perspective or "view," in the *IEEE 1471* sense, to describe rationale and architectural knowledge. The traditional gap between different artifacts of the software engineering process has shown the need to effectively and precisely capture and represent design decisions and their underlying rationale for later use, thus avoiding knowledge vaporization.

We also believe that key architectural design decisions should be recorded and documented; in contrast, it's not worth the effort to capture and maintain all the microdecisions that happen along a software system's life. One adoption barrier for capturing design decisions is the intrusiveness of many of the processes involved, as they're not fully integrated into current software engineering practice. So, tools such as those mentioned in the sidebar "Tools Supporting Design Rationale" must be improved, adapted, and better integrated to avoid

About the Authors

Philippe Kruchten is a professor of software engineering in the University of British Columbia's Department of Electrical and Computer Engineering. He spent more than 30 years in industry, working mostly with large software-intensive systems design in telecommunication, defense, aerospace, and transportation domains. Kruchten directed the development of the Rational Unified Process from 1995 to 2003. His research interests are software architecture, particularly architectural decisions and the decision process, and software engineering processes, particularly the application of agile processes in large, globally distributed teams. He received his doctorate in computer science from the French Institute of Telecommunications. He's a senior member of the IEEE Computer Society, the founder of Agile Vancouver, and a Professional Engineer. Contact him at pbk@ece.ubc.ca.

Rafael Capilla is an assistant professor of software engineering in the Computer Science Department at Universidad Rey Juan Carlos. His research interests include software architectures, product-line engineering, software variability, and Internet technologies. He received his PhD in computer science from Universidad Rey Juan Carlos. Capilla is a member of the IEEE Computer Society. Contact him at rafael.capilla@urjc.es.

Juan Carlos Dueñas is a professor in the Telecommunications School and currently the deputy director of the Department of Telematics Engineering at Universidad Politécnica de Madrid. He received his PhD in telecommunications from the same university. His research focuses on Internet services, service-oriented architectures, software architecture, software engineering, and system evolution. Dueñas coedited *Software Product Lines: Research Issues in Engineering and Management* (Springer, 2006) and is a member of the IEEE. Contact him at jcduenas@dit.upm.es.

duplicate efforts in capturing design decisions. They should also be used to facilitate the gradual introduction of new activities dealing with design rationale, some of which relate to distributed-team decision making.

There's often not much difference between the software requirements or description of a well-known design pattern and the explicit representation of a design decision. In many cases, a design decision constitutes a replica of the requirement that motivated that decision. As a result, the effort to capture such decisions is considered duplicated, because users of such tools often record the same data. So, appropriate mechanisms should be provided to avoid recording the same information as well as to streamline the capturing effort. These mechanisms must be based on stronger tracing and duplication-detection techniques.

The key goal of our current research is to highlight the importance and impact of design rationale in software architecture activities in particular, and in software engineering from a broader perspective. What will a fourth epiphany bring? Despite the challenges of capturing the design rationale, the introduction of documented design decisions will bring better ways to build and understand our software systems. Software architects and developers will also see the benefits of considering decisions first-class entities, and they will pursue better integration with other software engineering artifacts. Hopefully, design decisions and design rationale will be recognized in the upcoming ISO/IEC 42010 standard.

References

1. W.W. Gibbs, "Software's Chronic Crisis," *Scientific American*, vol. 271, Sept. 1994, pp. 72–81.
2. P. Kruchten, H. Obbink, and J. Stafford, "The Past, Present, and Future of Software Architecture," *IEEE Software*, vol. 23, no. 2, 2006, pp. 22–30.
3. D.E. Perry and A.L. Wolf, "Foundations for the Study of Software Architecture," *ACM Software Eng. Notes*, vol. 17, no. 4, 1992, pp. 40–52.
4. P.. Kruchten, *The Rational Unified Process—An Introduction*, 3rd ed., Addison-Wesley, 2003.
5. *IEEE Std. 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems*, IEEE, 2000.
6. P. Kruchten, "The 4+1 View Model of Architecture," *IEEE Software*, vol. 12, no. 6, 1995, pp. 45–50.
7. C. Hofmeister, R. Nord, and D. Soni, *Applied Software Architecture*, Addison-Wesley, 1999.
8. C. Hofmeister et al., "A General Model of Software Architecture Design Derived from Five Industrial Approaches," *J. Systems and Software*, vol. 80, no. 1, 2007, pp. 106–126.
9. P. Clements et al., *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2002.
10. N. Rozanski and E. Woods, *Software Systems Architecture*, Addison-Wesley, 2005.
11. I. Rus and M. Lindvall, "Knowledge Management in Software Engineering," *IEEE Software*, vol. 19, no. 3, 2002, pp. 26–38.
12. J. Bosch, "Software Architecture: The Next Step," *Proc. 1st European Workshop Software Architecture* (EWSA 04), LNCS 3047, Springer, 2004, pp. 194–199.
13. P. Kruchten, P. Lago, and H. van Vliet, "Building Up and Reasoning about Architectural Knowledge," *Proc. 2nd Int'l Conf. Quality of Software Architectures* (QoSA 06), LNCS 4214, Springer, 2006, pp. 43–58.
14. J. Tyree and A. Akerman, "Architecture Decisions: Demystifying Architecture," *IEEE Software*, vol. 22, no. 2, 2005, pp. 19–27.
15. R. Capilla, F. Nava, and J.C. Dueñas, "Modeling and Documenting the Evolution of Architectural Design Decisions," *Proc. 2nd Workshop Sharing and Reusing Architectural Knowledge Architecture, Rationale, and Design Intent*, IEEE CS Press, 2007, p. 9.
16. T. Käkölä and J.C. Dueñas, eds., *Software Product Lines—Research Issues in Engineering and Management*, Springer, 2006.
17. A.H. Dutoit et al., eds., *Rationale Management in Software Engineering*, Springer, 2006.
18. P. Lago and P. Avgeriou, "First ACM Workshop on Sharing and Reusing Architectural Knowledge (Shark)," *ACM SIGSOFT Software Eng. Notes*, vol. 31, no. 5, 2006, pp 32–36.
19. D. Falessi, R. Capilla, and G. Cantone, "A Value-Based Approach for Documenting Design Decisions Rationale: A Replicated Experiment," *Proc. 3rd Int'l Workshop Sharing and Reusing Architectural Knowledge* (Shark 08), ACM Press, 2008, pp. 63–70.
20. R. Capilla, F. Nava, and R. Carrillo, "Effort Estimation in Capturing Architectural Knowledge," *Proc. 23rd IEEE/ACM Int'l Conf. Automated Software Eng.*, IEEE Press, 2008, pp. 208–217.
21. J.C. Dueñas and R. Capilla, "The Decision View of Software Architecture," *Proc. 2nd European Workshop Software Architecture* (EWSA 05), LNCS 3047, Springer, 2005, pp. 222–230.