The Decision View's Role in Software Architecture Practice

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

This is a journey of discovery from software architecture representation to architectural methods, to design decisions, to end at 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

> *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*

0740-7459/09/$25.00 © 2009 IEEE

Standard 1471-2000 Recommended Practice for 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 its 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 and 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.

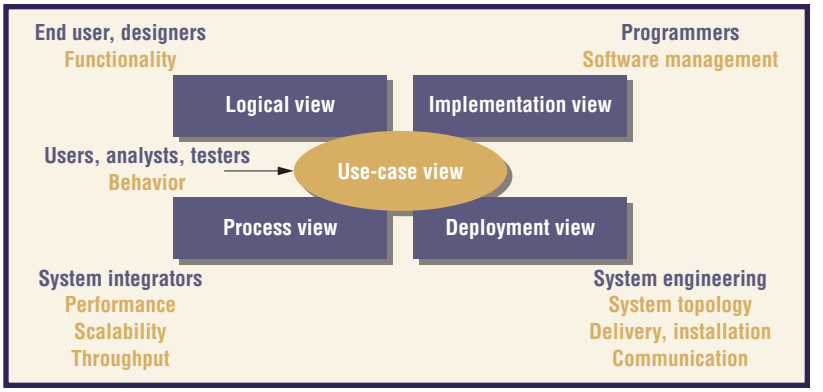

Figure 1. The "4+1" architecture view model.[6] Four views describe the design concerns of different stakeholders. A use-case view overlaps the others and relates the design to its context and its business goals.

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 essential phases of the architecting activity but are performed in different ways.

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

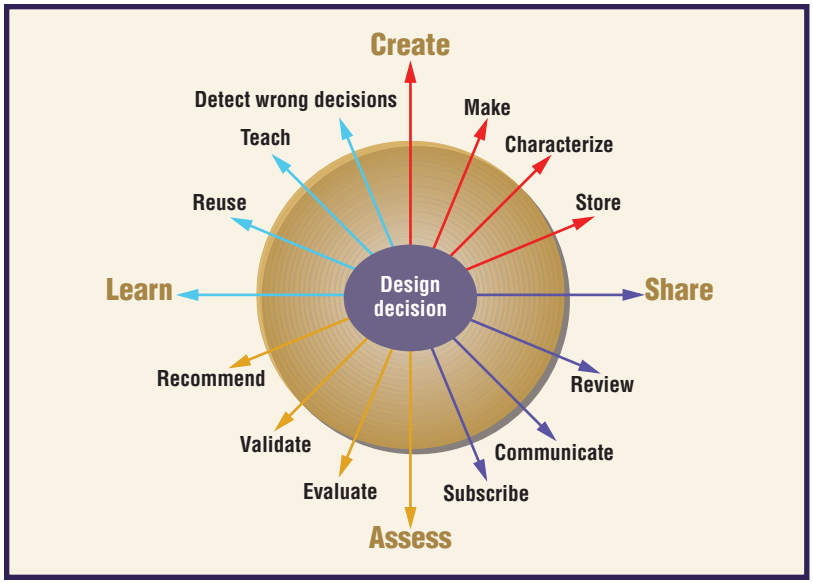

Figure 2. The four main activities—Create, Share, Assess, and Learn—and the subactivities involved in the creation and use of design decisions and design rationale. Each of the four colors shown indicates a main category (for example, "Create") and its related, smaller subactivities (in this case, "Make,"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

under architects' guidance, elicit these decisions, but 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 a first version of the architecture is created, the design can be communicated to other stakeholders and, for instance, the status of the architecture can be reviewed. During maintenance, decisions might become relevant for the current architecting team to evaluate and to provide recommendations to determine whether the decisions 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 //okay?// 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.

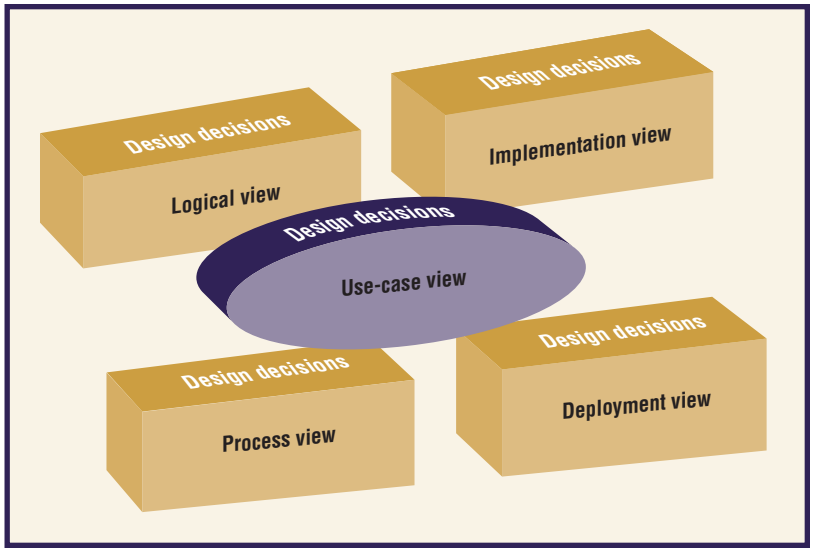

Figure 3. The "decision view" embedded in the 4+1 view model. This new perspective superimposes the design rationale that underlies and motivates the selection of concrete design options.

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, particularly in successive iterations of the system as it evolves.

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; most of them rely on the development of scenarios, their projection against several candidate architectures, and the addition of information to the architectural components. Then the architect 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, as shown 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 the designs' evolution but also the evolution and maintenance of the decisions base itself. This issue often emerges during reviews, where major changes affect the design.

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 **to discuss //???//** 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.

Like other key 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 listed in Figure 2, as they're not fully integrated into current software engineering practice. So, tools such as those mentioned in the sidebar "Tools Supporting Design Rationale"

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 in 1995–2003. 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. He received his PhD in computer science from the same university. His research interests include software architectures, product line engineering, software variability, and Internet technologies. Capilla is a member of the IEEE Computer Society. Contact him at rafael.capilla@urjc.es.

Juan C. 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 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.

must be improved, adapted, and better integrated to avoid 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*, September 1994, vol. 271, 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 Valued-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.

Es lícita la inclusión en una obra propia de fragmentos de otras ajenas de naturaleza escrita, sonora o audiovisual, así como la de obras aisladas de carácter plástico o fotográfico figurativo, siempre que se trate de obras ya divulgadas y su inclusión se realice a título de cita o para su análisis, comentario o juicio crítico. Tal utilización solo podrá realizarse con fines docentes o de investigación, en la medida justificada por el fin de esa incorporación e indicando la fuente y el nombre del autor de la obra utilizada.

Reflective Approach for Software Design Decision Making

Conference Paper · April 2016

DOI: 10.1109/QRASA.2016.8

CITATIONS
9

READS
270

4 authors:

Maryam Razavian
Eindhoven University of Technology
37 PUBLICATIONS **375** CITATIONS
SEE PROFILE

Antony Tang
Swinburne University of Technology and VU University Amsterdam
77 PUBLICATIONS **2,093** CITATIONS
SEE PROFILE

Rafael Capilla
King Juan Carlos University
131 PUBLICATIONS **1,803** CITATIONS
SEE PROFILE

Patricia Lago
Vrije Universiteit Amsterdam, Amsterdam, Netherlands
263 PUBLICATIONS **3,837** CITATIONS
SEE PROFILE

Reflective Approach for Software Design Decision Making

Maryam Razavian
Eindhoven University of Technology, The Netherlands
Email: m.razavian@tue.nl

Antony Tang
Swinburne University of Technology, Australia
Email: atang@swin.edu.au

Rafael Capilla
Universidad Rey Juan Carlos, Spain
Email: rafael.capilla@urjc.es

Patricia Lago
Vrije Universiteit Amsterdam, The Netherlands
Email: p.lago@vu.nl

***Abstract*—Good software design practice is difficult to define and teach. Despite the many software design methods and processes that are available, the quality of software design relies on human factors. We notice from literature and our own experiments that some of these factors concern design reasoning and reflection. In this paper, we propose a reflective approach to software design decision making. The approach is built upon Two-Minds model and is enabled by a set of problem-generic reflective questions. We illustrate its usefulness in design sessions with an example taken from preliminary experimentation.**

Keywords*—*Design Decision Making, Reflection, Behavioral Software Engineering

I. Introduction

It is well recognized in the software architecture community that design rationale and decision making are important aspects in software architecture design [1], [2]. There are many models that deal with the artifacts such as decision models or design rationale elements. However, these models and elements lack considerations of human factors in design, such as cognitive biases, proper design reasoning and communication, and reflection; all of which can affect decision making [3]. For instance, a designer who is biased towards one solution can put misleading emphasis on its rationale. As such, having rationale and decision models do not necessarily produce good software architecture design.

Human issues are a well known problem in design decision making [4]. It has been shown that people generally have a pattern of deviation in judgment, due to biases such as subjective representativeness of probability, bounded rationality, satisficing [5] and so on [6]. As an initial attempt to overcome some of the mentioned human issues, we discuss a design decision making approach that treats design reasoning and reflection as first class elements.

Novice software designers are trained to use design and development methodologies such as object-oriented analysis and design, RUP, or SCRUM. There is however few methodologies that address *design thinking*; i.e, how software designers should think when designing. Design thinking has many perspectives [7]. In this paper, we focus on two perspectives: design reasoning and reflection.

Design reasoning is a process which helps a software designer gather the right information, exploring relevant problems and synthesizing potential solutions. Studies such as the twin peak model [8], various decision rationale models [1] and design reasoning techniques [9] are available, but they do not directly challenge the thinking behind such design reasoning.

Reflection helps to challenge the thinking behind design reasoning. It can help to overcome human design thinking issues such as cognitive biases, satisficing behavior and lack of knowledge. Reflection helps to check and remind a designer if appropriate information is gathered, relevant problems are identified, and solution options and decisions are carefully considered.

Existing software design approaches provide mechanisms to carry out design steps, without any consideration how to reason when making software design decisions. To fill this gap, this paper takes the first crucial step by proposing an approach that facilitates *reflection on design decision making*. The approach is built upon the *Two-Minds* model for design thinking, presented in [10]. Two-Minds, captures the interplay of modes of thinking for design thinking at two levels: (a) the design reasoning level called *Mind 1*; and (b) reflection level called *Mind 2*. While the Two-Minds model provides the structure of design thinking, in this work we focus on the *reflective questions* that steer software design decision making. Our proposed reflective questions target two very different decision making approaches: *rational* and *naturalistic* [11]. Naturalistic decision making is based mainly on intuition and experience. Rational decision making relies on careful reasoning and argumentation. With these two basic decision making approaches, we explore our reflection method.

Section II explains the theoretical underpinnings of our proposed reflective approach. Section III further discusses the rational and naturalistic decision making approaches, in the context of software design, as well as how our reflective approach can steer each of the two. We experimented with our reflection method using novice designers. It was found that [10] those who use a reflective mind generally produced better quality work. In this paper, we illustrate how reflective questions work for novice in the design decision making (Section IV). We conclude with a discussion of promising future research directions.

II. Background: Software Design Thinking Model

Software design has often been seen as a problem solving exercise—that is, analyzing problems, finding solution options, and making decisions. The process of challenging the thinking

behind such problem solving is not explored. For instance, are there any implicit assumptions behind a design problem? Are risks considered when a solution is proposed? How does one decide if a requirement has priority over another requirement? or if we are tackling the real problem? Are cognitive biases involved in the problem solving? Facing such issues, we need better ways to deal with the assumption that all designs can come naturally without careful reasoning. But first we need a model to represent design thinking.

We present a model for *Software Design Thinking*, called Two-Minds, in [10] (see Fig. 1). Two-Minds reflects our theory that software design thinking comprises the following modes of thinking: (i) Mind 1 is the *design reasoning mind* with a *problem solving mindset*; (ii) Mind 2 is the *reflective mind* with a *feedback mindset*. Mind 1 is about design argumentation, whereas Mind 2 is about conscious questioning and reflection on how well we reason and argue with a design.

A. Mind 1: Design Reasoning

As illustrated in Fig. 1, Mind 1 carries out four generic activities: *Identifying Relevant Context and Requirements*, *Formulating and Structuring Design Problems*, *Creating Solution Options*, and evaluating the trade-offs of the potential solutions to *Make Decisions*. Having the right requirements and the relevant context can help to frame the right design problems to solve [9]. As problems and solutions are often intertwined, their explorations are also intertwining [12]. Accordingly, Mind 1 bidirectional arrows in our model capture such co-evolving exploration process in design reasoning.

Two points should be noted about Mind 1. First, although Mind 1 emphasizes the use of the four key activities to reason with a design it does not suggest the order of the activities. The process of applying this model is incremental and iterative. A design decision can spark new design problems, leading to new design options and context. Second, covering all the four activities is important for new and unfamiliar problems. As designers, if we are familiar with the problem and the solution, we might find it quite easy to solve that problem relying on our experience—in such a case we might skip some of the four reasoning steps when making a decision. However, if the problem is new and unfamiliar, we need to gather relevant requirements or premises, contemplate what problems to solve, and potential solutions for satisfying a set of requirements and context [13]—in such a case we are following a process involving all four activities of Mind 1.

Let us consider a couple of examples of Mind 1 reasoning. (1) To reason on which database to use, a designer asks "Should we use PostgreSQL? Is it better to use JSON or SOAP to transfer data?". (2) To develop a required API, a designer asks "Is it better to use JSON or XML to transfer data?". These design topics involve reasoning based on the technical know-how and the context of a design.

B. Mind 2: Reflection

Mind 2 is about challenging designers' thinking and reasoning. Just as any kind of reflection, it is difficult to *step out* from of the current mindset and consider problem-solving from a different level of abstraction. A reflective mind challenges one's problem solving process, which does not consider technical design details—it is about *how well to reason*.

As depicted in Fig. 1, our model includes four key areas of design reflection (Mind 2).

- *Reflect on the Contexts and Requirements.* This reflection challenges if our reasoning is based on relevant and adequate context and requirements. For instance, if certain context and requirements are relevant to a design argumentation or not; or if we have accurately described certain requirements (e.g., performance) in our argumentation.
- *Reflect on Design Problems* challenges if in our design reasoning the design problems have been well articulated. For instance, design problems can be unclear, imprecise and poorly articulated. Reflection is aimed at challenging these situations [14].
- *Reflect on Design Solutions.* It challenges how design solutions are arrived at. For example, sometimes designers do not provide solution options even when they exist; or a solution does not address any requirements [15]. In these cases, reflection helps to challenges the quality and the appropriateness of a how a solution is arrived at.
- *Reflect on Design Decisions.* It challenges whether the reasoning behind a design decision is sound and valid. For instance, if during the problem-solving the pros and cons of each solution options are considered.

To illustrate some examples of reflective mind compared to reasoning mind, we refer to the reasoning examples from Mind 1. A reflective Mind 2 would not only ask about what to use, but it would ask about the criteria that dictate the decision. Such as "What are some of the criteria that affects the selection of a database? performance? portability?" or "What is the criteria for a better protocol? payload overhead? language support?" The reflective Mind 2 in these examples help a designer to check his thinking and reflect on design decisions.

III. How Reflective Questions Steer Software Decision Making

This section presents our reflective approach to software design decision making. The approach is built upon Two-Minds model and is enabled by a set of problem-generic reflective questions.

A. Reflective Questions for Realizing Two-Minds

We propose *reflective questions* as a technique to trigger reflection on design reasoning. As illustrated in Fig. 1, reflective questions are meant to prompt moving from each of the activities of the reasoning mind (circles in the bottom of Fig. 1) to the reflective mind (circles on top of Fig. 1). For example, when *identifying relevant context and requirements* a reflective question such as "what else do we need to know?" would trigger the designer to *reflect on context and requirement*. Reflective questions can be applied to *context and requirements* (e.g., what else do I need to know?), *design problems* (e.g., what is the problem?), *design solutions* (e.g.,

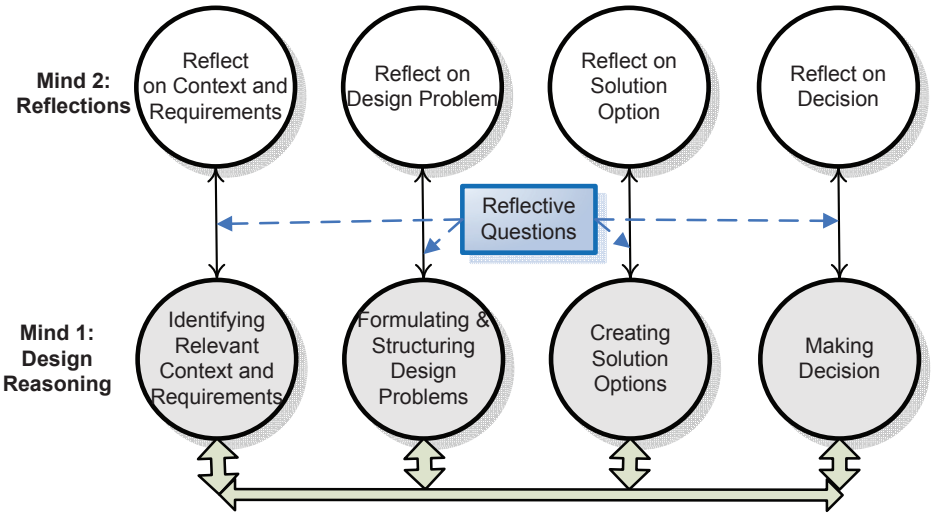

Fig. 1: Software Design Thinking Model - Mind 1 and Mind 2 [10]

do these solutions solve my problems), and one's potential biases in the evaluation of a decision (i.e. is the trade-off in the decision fair?). So what reflective question should one ask to challenge the design reasoning? Reflective questions can be applied to *context and requirements* (e.g., what else do I need to know?), *design problems* (e.g., what is the problem?), *design solutions* (e.g., do these solutions solve my problems?), and one's potential biases in the evaluation of a decision (i.e. is the trade-off in the decision fair?).

In our view reflective questions should pursue three goals: (i) provide a common ground to challenge design reasoning, (ii) be prevalent to software design endeavors and cover what we already address in problem solving, i.e., context, designs issues, design options, risks, and decisions, and (iii) be in-line with how designers work.

With these goals in mind, we defined a set of reflective questions (see Table I), derived from design reasoning techniques and customized for each of the design reasoning activities of our model. Each question challenges the generic design reasoning activities (referred to in the columns of Table I), while the rows show the design reasoning techniques the question belongs to, i.e., trade-off analysis, risk analysis, assumption analysis, constraint analysis and problem analysis. A summary of these techniques is reported in [9].

B. Prompting Rational and/or Naturalistic Decision Making

The reflective questions are meant to steer software architecture decision making. Decision making research, spanning an extensive range of disciplines such as business, economics, and psychology, have been focusing on various approaches of design decision making, ranging from *rational* to *naturalistic* [11].

Rational Decision Making (RDM) relies entirely on conscious understanding of all the design issues, analysis, and reasoning. RDM is characterized by systematic exploration of alternative options, rating options based on predefined set of criteria, resulting in logical arguments in favor or against those options [16]. This kind of decision making places a heavy load on cognition. *Naturalistic Decision Making* (NDM), provides a rather different account of how designers make decisions: through expertise and experience, designers learn to recognize situations and match them with a set of solutions that they know would work; if the match is not so strong the designers stimulate how actual solution might play out in the new situation and make the necessary modifications. This way of decision making is simpler and faster, and it has the added advantage that decisions are taken quickly [11].

RDM *prescribes* a rational way of design decision making. In NDM, a designer follows the perception of the relative *desirability* of the available options [17]. By attaining that understanding, a designer, is able to identify the most *robust option*, those that are more probable to turn out favorably under widest range of possible options. Robust options are in contrast to *optimal options* in RDM, options that score highest based on a set of predefined criteria.

The Two-Minds model examines a phenomenon common to both RDM and NDM: being aware of design context, issues, and options. Although RDM and NDM represent different approaches to decision making, there is no inconsistency between the role of reflection in the two: *questioning and challenging the way decisions are made*. Two-Minds model is effectively orthogonal to RDM and NDM design decision making: Mind 1 and Mind 2 encapsulate common activities in each of RDM and NDM, however, with different focuses: RDM emphasizes *accuracy* and *completeness*, while NDM emphasizes *expertise* and *complexity* [11]. More specifically, for each of Mind 1 activities (see bottom of Fig. 1):

- When *Identifying Context and Requirements* RDM requires that the context is defined *accurately* and *sufficiently*. NDM, however, focuses on understanding

TABLE I: Reflective questions used by Mind 2 to reflect on Mind 1 [10]

		Design Reasoning Activities			
		Design Contexts and Requirements	**Design Problems**	**Design Solutions**	**Decision Making**
Design Reasoning Techniques	**Assumption Analysis**	Q1. What assumptions are made?	Q2. Do the assumptions affect the design problem?	Q3. Do the assumptions affect the solution option?	Q4. Is an assumption acceptable in a decision?
	Risk Analysis	Q5. What are the risks that certain events would happen?	Q6. How do the risks cause design problems?	Q7. How do the risks affect the viability of a solution?	Q8. Is the risk of a decision acceptable? What can be done to mitigate the risks?
	Constraint Analysis	Q9. What are the constraints imposed by the context?	Q10. How do the constraints cause design problems?	Q11. How do the constraints limit the solution options?	Q12. Can any constraints be relaxed when making a decision?
	Problem Analysis	Q13. What are the context and the requirements of this system? What does this context mean?	Q14. What are the design problems? Which are the important problems that need to be solved? What does this problem mean?	Q15. What potential solutions can solve this problem?	Q16. Are there other problems to follow up in this decision?
	Tradeoff Analysis	Q17. What context can be compromised?	Q18. Can a problem be framed differently?	Q19. What are the solution options? Can a solution option be compromised?	Q20. Are the pros and cons of each solution treated fairly? What is an optimal solution after tradeoff?

the *complexity* of the context (e.g., contradicting goals or assumptions) as well as available *expertise* (e.g., skills and experiences of designers).

- For *Formulating & Structuring Design Problems* RDM focuses on *accurate* formulation of *all relevant* design issues, while NDM emphasizes *critical design issues* and *plausible goals*.
- When *Creating Solution Options* RDM aims at exploring the complete list of options, while NDM focuses on *workable* options, based on past experiences or what experts foresee as being *plausible*.
- For *Making Decisions*, RDM evaluate the options based on a set of *known* assumptions, constraints and qualities, in order to select the *optimal option*. NDM, however, focuses on evaluation of existing options in complex scenarios, in order to choose the *robust option*.

As noted, what Mind 2 does, i.e., challenging the four reasoning activities, is common to both RDM and NDM approaches. The differences between RDM and NDM lies in "what" needs to be challenged. Hence, the reflective questions should register the differences between each of Mind 1 activities in RDM and NDM, as discussed above. We propose a paired down list of reflective questions for each of RDM and NDM approaches (see Table II and Table III). Those questions are defined based on the different focus of each of the RDM and NDM approaches as described above and customized for each of the design reasoning activities (presented in brackets in Table II and Table III).

On one hand, the seven RDM-inspired reflective questions (see Table II) help to structure the design problems, generate alternative options, and evaluate them based on set of known assumptions, constraints and qualities. For instance, if a designer thinks that there are certain risks in the design, then he/she can pose the RDM-inspired question of *"What can potentially go wrong with... ?"*(Q4 in Table II). Consider the case where in a design session the 'circular buffer' is selected as a way to realize a FIFO queuing mechanism. A question like *"What can go wrong with the choice of using a circular buffer?"* can help formulating and structuring the consequent design problems, such as, *"expanding a circular buffer requires shifting memory, which is comparatively costly"*. As an alternative option a designer could argue *"for arbitrarily expanding queues, a linked list approach may be preferred instead"*.

On the other hand, the seven NDM-inspired reflective questions (see Table III) can help to characterize the context and requirements, recognizing common problems, representing design solutions and their potential outcomes, and providing means to think in new ways about the underlying reasons for variations in potential outcomes and enabling addressing hidden complexity. For instance, and NDM-inspired questions like *"Is this solution workable for . . . "*. This reflective question enables designers to recognize situations and match them with a set of solutions that they know would work. Consider the case where in a design session 'Amazon S3 Web Services' is selected as a solution for storage of data objects (e.g., hotels info). A question like *"Does S3 work for querying detailed info like hotels with swimming pool?"* (i.e., customized based on Q5 in Table III) can help identify the necessary modifications. As an answer a designer might argue: *"We need to introduce a certain file naming scheme like hotels with/without swimming pool"*.

It should be noted that, extreme forms of pure rational or naturalistic decision making are unlikely to happen in software design development. In practice, software designers use a combination of both rational and naturalistic approaches [17]. This means that in any design session, a combination of RDM- and NDM-inspired questions can be used. While the reflective questions enable prompting each of the RDM and/or NDM approaches, when/how RDM and NDM should be used is yet unknown. One possible factor is designer's expertise, discussed in the following.

C. Steering Decision Making based on Designer's Expertise

Much emphasis is placed on the role of expertise in decision making and significant difference of novice and experts.

TABLE II: RDM-inspired Reflective Questions

Reflective Questions based on RDM
Q1. What are the most important…? [Identifying Context and Requirements] *What are the most important requirements here?* *Which design consideration is more important to the user, A or B?*
Q2. How does that address…? [Identifying Context and Requirements] *How does that address the requirement of having to authenticate individual transaction?* *How do we deal with the fact that no programmers in the team know JQuery?*
Q3. What is your reason for…? [Formulating Design Problems] *What is the reason for selecting service oriented architecture?* *Why do you use asynchronous communication protocol?*
Q4. What can potentially go wrong with…? [Formulating Design Problems] *What can go wrong with the choice of using a circular buffer?* *What are risks with the confidentiality of user data?*
Q5. What problems are you solving with…? [Formulating Design Problems] *What problems are you solving with migration to the cloud?* *What are the design problems to consider in designing a controller for a self-guiding robot?*
Q6. What are the alternatives to…? [Creating Solution Options] *What are the alternatives to using the cloud?* *Can you use another data structure?*
Q7. What are the potential limitations to…? [Making Decisions] *What are the potential limitations to choosing MySQL Lite?* *How would the assumption on the transaction rate of MySQL Lite influence the performance of your system?*

TABLE III: NDM-inspired Reflective Questions

Reflective questions based on NDM
Q1. Is this context familiar where…? [Identifying Context and Requirements] *Have we faced this situation before?* *What are the commonalities between the current situation and the one experienced before?*
Q2. Have we addressed this goal before when…? [Identifying Context and Requirements] *Have we addressed integration of silo systems before?* *How do we deal with security this time?*
Q3. What goals are plausible when…? [Formulating Design Problems] *Can we achieve flexibility when selecting service oriented architecture?* *What was our gain in using asynchronous communication protocol before?*
Q4. Have we faced similar problems in…? [Formulating Design Problems] *Did we face the data interoperability problem in project…?* *What was the performance problem in…?*
Q5. Is this solution workable for…? [Creating Solution Options] *Does Amazon Web Services work for our storage problem?* *What is the typical solution for avoiding data leakage?*
Q6. What will the outcome look like when…? [Making Decisions] *What will the performance be if we use MySQL Lite?* *How will the controller component look like in a self-guiding robot?*
Q7. Does the outcome fulfill…? [Making Decisions] *Did using cloud fulfill our energy efficiency expectations?* *Can we need to modify our security solution?*

Although NDM captures what experienced designers do, it does not specify how we can determine whether software designers are experts. Some argue that expertise is a continuum, and there are boundaries and voids within all designer's expertise whether they have 5 or 25 years of experience [18]. Software designers make decisions across domains where there are disagreements about what the appropriate action should be. When problems are familiar and well defined, designers are able to map the situation to the one experienced previously and quickly come to conclusion regarding robust options, without having to resort to a rational approach [17]. In such cases, NDM-inspired reflective questions like "*what did I do last time that worked?*", could help to steer decision making. However, when problems are new and uncertain and the designers have limited experience, it is difficult for designers to envision which options are more likely to be desirable and result in a successful decision. To steer such decision making, we conjecture that use of RDM-inspired reflective questions like such as "*based on what criteria the options should be compared?*" can better facilitate decision making. Reflective questions proposed in Table II and Table III are a pragmatic way support both RDM and NDM approaches depending on

a designer's design approach and expertise.

IV. Steering Rational Decision Making of Novice Designers

Good design decision making is an acquired skill. Novice or inexperienced software designer, no matter how relatively skilled they are, simply will not have enough collection of known and experienced situations, problems or working solutions to pattern match current situations or problems. Thus a novice's use of NDM is limited. For this reason, we explored the use of the RDM-inspired questions (see Table II) with novice software designers, in the context of an experiment [10]. In what follows we explain how those reflective questions steered the *rational decision making* of novice designers.

Before the design sessions, we taught students the generic design reasoning activities (See bottom of Fig. 1) and design reasoning techniques such as QOC [19] (Mind 1). However, we did not teach or inform the students about reflection or reflective questions (Mind 2). We participated in the design sessions of student teams and posed reflective questions from Table II. In a way, we played the role of "reflection advocate", a designer in the team who observes the thinking behind the design process and when necessary triggers the reflective mind. As reflective advocates we followed one rule: we did not provide design suggestions or give design solutions, we only asked questions.

The design dialogue shown in Table IV captures part of a design session in a design team. Design dialogue represents the thinking behind a design—it represents the identifiable states of Mind 1 and Mind 2 in group design situation. We used the RDM-inspired reflective questions and contextualized them based on our conversation to trigger novice designers to use the reflective mind (Mind 2). Throughout the design session when the discussion was deviating from rational approach, we triggered Mind 2. For instance, when the context was not explored sufficiently, or argumentation was not carefully constructed, and options were not scored based on a set of criteria, we posed a reflective question.

We use the design dialogue shown in Table IV to illustrate how reflective questions help rational decision making. The dialogue starts with the students discussing their envisioned solution without discussing the problem. As the reflection advocate, we observed that there is insufficient problem understanding. Rational approach, however, demands understanding, formulating and structuring the problem before selecting a solution. Thus, to prompt students to reflect on design issues, we posed a reflective question (line 1 in Table IV). As an answer to the reflective question, the student raised the issue of security (line 2). The student used the word "security" vaguely without a clear notion of what it means to their system. This issue, however, was vaguely formulated. Rational decision making, conversely, aims at accuracy of information, allowing precise analysis of problem and solutions. Noticing this, we sought further clarification of the context to see if the students understood why confidentiality was relevant in this situation. By asking how the students defined confidentiality, we prompted them to reflect on the design issue (line 3); the answer given was superficial and no further information was given in this answer (line 4). At this point, we made another attempt asking the student to phrase the design issues in the form of a question (line 5); in answering this question the issue became more articulated (in line 6). Interestingly, reflective questions had triggered students to find other design issues (line 7).

We see that the three reflective questions helped the students to some extent fulfill two of the key requirements of rational decision making approach: *accuracy* and *completeness* of design issues [16]. It helped the to step out and *Reflect on Design Problem*. Such reflection allowed them to frame their design problems from a generic confidentiality issue to *specific* design issues that needed to be solved. In other words, it helped them to improve the *Formulating & Structuring Design Problems* in their reasoning and make it more *accurate* (accuracy of design issues). It also led them to identify additional relevant design issues (completeness of design issues). Ultimately, rational decision making approach advocates that high quality problem formulation is a foundation of good design.

In a nutshell, our initial findings suggest that it is worthwhile to inject reflective questions in the software design teams to challenge rational decision making in a more structured way.

V. Discussion and Outlook

Studies in design science tell us that people can make biased design decisions. However, software engineering design approaches and models have barely addressed how human factors influence design. One of these human factors is the inability of a software designer to reflect on one's reasoning. By treating this human factor as a first class element, we proposed a reflective approach to software design decision making. We have shown in a previous work that reflection has a positive impact on the quality of design decision making. However, the systematic practice of reflection and reasoning requires more empirical studies and experiments to understand how reflection leads to better software design decisions: What questions to ask and when to ask them? What reflective questions are effective? And under what circumstances? Does personality play a role? This knowledge will allow us to improve software design, in three ways:

a) Towards complementing RDM with NDM: Before naturalistic decision making (NDM) was introduced, decision making researchers primarily focused on rational decision making (RDM), that is systematic and thorough identification of decisions among alternatives. With the rise of naturalistic decision making and the research on biases (e.g., Kahneman et al. [4]) it was known that designers do not always adhere to the principles of RDM; designers often rely on their intuition rather than reasoning. They mostly do not generate alternative options or evaluate the options based on predefined set of criteria. However, in the software architecture field researchers have mainly focused on a rationalistic approach on software architecture decision making. Most of the work focused on generating and documenting design options, decisions, and their rationale. However, extreme forms of pure rational or naturalistic decision making are unlikely to appear in any software design and development endeavor. In practice, software designers use a combination of both rational and naturalistic approaches [17]. For instance, intuition, a key element of

TABLE IV: An Example of Reflective Approach in Design Sessions

Design Dialogue	Interpretation
0. Student 3: We should design a service for admission of students. The students from all over the world should be able to enter the required information.	The students had a discussion about the type of solution they envision without thinking about the problem.
1. Reflection Advocate: What are your design issues for this business service? (**Q5** in Table II)	This was a reflective question to prompt the students to identify the design issues before thinking about solutions.
2. Student 3: The most important one is security, because we need the data to be confidential.	The student identified on security as a design issue, and tried to justify why it was an issue
3. Reflection Advocate: What does confidentiality mean for your domain? (**Q3**)	We sought further clarification of the context to make student reflect on why confidentiality was relevant in this situation and provide an accurate definition.
4. Student 3: It means that the data of student is only visible to authorized people.	The answer given was superficial and no further information was given in this answer.
5. Reflection Advocate: Can you rephrase this issue as a question? (**Q2**)	This reflective question prompted the students to look at the design issue from another perspective.
6. Student 3: How to make sure the data of students remain confidential?	One student added data confidentially as a design issue, providing more specific issues to be addressed.
7. Student 1: We also have another issue about authorization Let's say "how to authorize admission staff?"	Another student added user authorization as a design issue, a separate security issue but also specific to security.

naturalistic approach, can be traced in many design discourses and documents. Some elements of rationalistic approach, such as documenting the decision rationale, has been introduced in practice to support sound reasoning—though not always achieved—as desirable goal [20]. How the two should be balanced and combined, is still an open question.

b) Complexity and software designers in action: In the software architecture field (including our experiment with Two-Minds) we have been trying to identify when designers are suboptimal; when do they fail to make sound arguments and make optimal decisions. We conjecture that we should shift our focus on how software designers make decisions under tough circumstances such as limited time, uncertainty, and unstable conditions; how people use their experience to recognize a situation and generate fitting solutions rather than making a choice from a set of predefined options. NDM researchers in fields such as medicine [21], business [22], and economy [4], had already been studying these kinds of issues. There is an urgent need for research clarifying the problems and needs of software designers when dealing with complexity.

c) Integrating Two-Minds into software design practice: We argue that with the Two-Minds model software teams can systematically reflect on their design activities to achieve better results. To put Two-Minds model into practice, one can integrate it into existing design approaches such as agile methods. Being people-centric, agile methods rely on interaction and (for design activities) conversation about design among developers. This is an environment that naturally encourages argumentation and reflection about the thinking behind the design. In the future we plan to study how to realize such integration. Thus, while generally applicable, it seems especially promising to integrate the Two-Minds model in existing agile practices. This would help reflecting on design reasoning e.g., by introducing the role of "reflection advocate", a role similar to the role we played in the design sessions with students. The reflection advocate is an outsider in the design endeavor who often have different starting positions, challenge assumptions, and raise alternative interpretations of the problem.

REFERENCES

[1] D. Falessi, L. C. Briand, G. Cantone, R. Capilla, and P. Kruchten, "The value of design rationale information," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 3, p. 21, 2013.

[2] C. Zannier, M. Chiasson, and F. Maurer, "A model of design decision making based on empirical results of interviews with software designers," *Information and Software Technology*, vol. 49, no. 6, pp. 637–653, 2007, qualitative Software Engineering Research.

[3] H. van Vliet and A. Tang, "Decision making in software architecture," *Journal of Systems and Software*, pp. –, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216000157

[4] D. Kahneman, *Thinking, fast and slow*. Penguin, 2011.

[5] A. Tang and H. van Vliet, *Software Designers Satisfice*. Springer International Publishing, 2015, vol. 9278, ch. 9, pp. 105–120.

[6] M. Petre, A. van der Hoek, and A. Baker, "Editorial," *Design Studies*, vol. 31, no. 6, pp. 533 – 544, 2010, special Issue Studying Professional Software Design.

[7] K. Dorst, "The core of "design thinking" and its application," *Design Studies*, vol. 32, no. 6, pp. 521 – 532, 2011, interpreting Design Thinking.

[8] B. Nuseibeh, "Weaving Together Requirements and Architecture," *IEEE Computer*, vol. 34, no. 3, pp. 115–119, March 2001.

[9] A. Tang and P. Lago, "Notes on design reasoning techniques (v1.4)," Swinburne University of Technology, Tech. Rep., 2010.

[10] M. Razavian, A. Tang, R. Capilla, and P. Lago, "In Two Minds: How Reflections Influence Software Architecture Design Thinking," *Journal of Software: Evolution and Process [In Press]*, 2016.

[11] G. Klein, "Naturalistic decision making," *Human Factors: The Journal of the Human Factors and Ergonomics Society*, vol. 50, no. 3, pp. 456–460, 2008.

[12] K. Dorst and N. Cross, "Creativity in the design space: co-evolution of problem-solution," *Design Studies*, vol. 22, no. 5, pp. 425–437, 2001.

[13] A. Tang and H. van Vliet, "Design strategy and software design effectiveness," *IEEE Software*, vol. Jan-Feb, pp. 51–55, 2012.

[14] K. Dorst, "Design problems and design paradoxes," *Design issues*, vol. 22, no. 3, pp. 4–17, 2006.

[15] A. Tang and M. F. Lau, "Software architecture review by association," *Journal of Systems and Software*, vol. 88, no. 0, pp. 87–101, 2014.

[16] D. Parnas and P. Clements, "A Rational Design Process: How and Why to Fake it," *IEEE Transactions on Software Engineering*, vol. 12, no. 2, pp. 251–257, 1986.

[17] M. S. Pfaff, G. L. Klein, J. L. Drury, S. P. Moon, Y. Liu, and S. O. Entezari, "Supporting complex decision making through option awareness," *Journal of Cognitive Engineering and Decision Making*, vol. 7, no. 2, pp. 155–178, 2013.

[18] R. S. Adams, S. R. Daly, L. M. Mann, and G. Dall'Alba, "Being a professional: Three lenses into design thinking, acting, and being," *Design Studies*, vol. 32, no. 6, pp. 588–607, 2011.

[19] A. Maclean, R. Young, V. Bellotti, and T. Moran, "Questions, Options and Criteria: Elements of Design Space Analysis," in *Design Rationale: Concepts, Techniques and Use*, T. Moran and J. Carroll, Eds. Lawrence Erlbaum Associates, 1996, ch. 3, pp. 53–106.

[20] M. Jackson, "Formalism and intuition in software engineering," in *Perspectives on the Future of Software Engineering*. Springer, 2013, pp. 335–347.

[21] A. S. Elstein, L. S. Shulman, and S. A. Sprafka, "Medical problem solving an analysis of clinical reasoning," 1978.

[22] D. J. Isenberg, *How senior managers think.* Open University Press, 1991.