The Journal of Systems and Software 88 (2014) 87–101

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Software architecture review by association☆

Antony Tang*, Man F. Lau

Faculty of Information and Communication Technologies, Swinburne University of Technology, Hawthorn 3122, Australia

ARTICLE INFO

Article history:
Received 11 March 2013
Received in revised form
26 September 2013
Accepted 26 September 2013
Available online 8 November 2013

Keywords:
Software architecture review
Design reasoning
Verification of software architecture

ABSTRACT

During the process of software design, software architects have their reasons to choose certain software components to address particular software requirements and constraints. However, existing software architecture review techniques often rely on the design reviewers' knowledge and experience, and perhaps using some checklists, to identify design gaps and issues, without questioning the reasoning behind the decisions made by the architects. In this paper, we approach design reviews from a design reasoning perspective. We propose to use an association-based review procedure to identify design issues by first associating all the relevant design concerns, problems and solutions systematically; and then verifying if the causal relationships between these design elements are valid. Using this procedure, we discovered new design issues in all three industrial cases, despite their internal architecture reviews and one of the three systems being operational. With the newly found design issues, we derive eight general design reasoning failure scenarios.

1. Introduction

Software architecture is one of the many important artifacts in software development. It is defined as "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution." (ISO/IEC, 2010), or it is a model of *elements*, *form* and *rationale* (Perry and Wolf, 1992) where (1) *elements* are processing, data or connecting elements, (2) *form* is defined in terms of the properties of the elements as well as their relationships, and (3) *rationale* is the constraints of the system that provide the underlying basis of the architecture. Basically, software architecture is an abstract model of a software system that aims to satisfy some software requirements.

As software systems are becoming increasingly complex and involving more and more stakeholders, it is widely acceptable to develop software architectures via iterations. During each iteration, the architecture is refined to address specific concerns and requirements. The word "iteration" here is used in a generic sense. On one hand, it may refer to different levels of abstractions in the architecture. For example, at the early stage of the design process, one of the main concerns is on the satisfaction of the software requirements. At a later stage, software architects may need to use quantifiable means to ensure that various hardware and software components can interact with each other to satisfy certain non-functional requirements such as system performance. On the other hand, it may refer to different viewpoints such as business, data and technology in the TOGAF (The Open Group, 2003). For example, the focus of the business viewpoint is on the satisfaction of the business requirements; whereas that of the technology viewpoint is whether the choice of technologies and networking infrastructure can together satisfy the software requirements.

During each design iteration, architectural design decisions are made and these decisions impact different aspects of an architecture design and future design decisions and choices. These decisions can be explicitly discussed or implicitly made. They can be documented or they may be tacit. They can come from one and, often, many designers. The design decisions with their assumptions, inter-relationships, reasoning and impacts are often intricate. These are important information to assuring a consistent and valid software architecture. Without them, reasoning flaws such as the following can happen:

1. the impacts and assumptions of a decision are not necessarily known to decision makers who deal with other parts of a system (Parnas and Weiss, 1987),
2. the goals, requirements and design concerns are unknown or not communicated to the people who need to know (Guindon et al., 1987; Brooks, 2010),
3. a decision maker is not aware of the implications of their decisions when they state a goal or a requirement (Guindon et al., 1987),

☆ This work was supported in part by a grant from the DCRG scheme, Faculty of Information and Communication Technologies, Swinburne University of Technology. We thank the three companies which supported our research. We also thank Ms. Alice Yip for her efforts in collecting and analyzing the specifications.

* Corresponding author. Tel.: +61 3 92148739.
E-mail addresses: atang@swin.edu.au (A. Tang), elau@swin.edu.au (M.F. Lau).

0164-1212/$ – see front matter © 2013 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2013.09.044

4. a decision maker is not aware of the implications of their design (Guindon and Curtis, 1988),
5. a chosen design creates conflicts in other parts of the system because not all constraints are integrated (Guindon and Curtis, 1988),
6. designers make biased and uninformed decisions (Stacy and MacMillan, 1995; Tang, 2011).

Since software architecture is developed through iterations and over time, it is customary to review the proposed architecture at the end of each iteration to ensure its validity. Such a review process is usually referred to as the *software architecture review* (Maranzano et al., 2005). Depending on the policies of the organizations, a software architecture review can be performed informally by some architects themselves or formally by quality assurance teams. Reviews aim to reduce the chance and the costs of correcting any major errors at later stages of development.

Despite the importance of design decision making, current software architecture review techniques are not effective in addressing design reasoning flaws. It is because general review techniques focus on architecture design artifacts instead of the logical design reasoning behind an architecture design.

Ideally, to address the aforementioned design reasoning flaws, software architects need to share "everything" related to design decision and reasoning so that architecture reviewers could track and trace those design decisions easily, and then reviewers would have the knowledge to help find design issues. In this paper, the term "design issue" is used in a generic sense. Whenever a causal relationship is illogical, it implies a potential issue in a design. Software architects usually use text documents with design diagrams (e.g. UML diagrams) to record and communicate their designs. If an important design assumption is not documented, it can be difficult for a reviewer to know that such an assumption exists, and relate its relevance to a design. It is also difficult to trace an implicit assumption back to a particular design artifact or requirements.

In view of these situations, we propose to review software architectures by analyzing the association between different design elements, hoping to identify (or, "re-identify") and verify those design assumptions, reasoning, decisions and tacit knowledge used by the architects in designing. Formally, we propose the concept of *Design Association Theory* which theorizes that a software architecture design is a network of associations of causal relationships between design elements in a meaningful way. The associations represent the causal relationships in a design, that is, software artifacts and the way they are constructed exist for good reasons. A design must have logical causes between a solution and the requirements and context that it serves. Loosely speaking, our approach helps reviewers to capture or "re-discover" those implicit and hidden reasoning and decisions made by software architects while designing. Once these "decisions" were represented as casual relationships, reviewers could then easily trace and track all these "decisions", and review them to see whether they are reasonable.

We have two contributions. Firstly, based on Design Association Theory, we describe a procedure for reviewing an architecture design to find design issues. This procedure is based on graphical representation of design associations. Secondly, our aim is to help architecture reviewers and designers identify missing design elements and systematically discover design issues through reasoning. From the design issues identified in our industrial case studies, we generalize them into eight different design reasoning failure scenarios. These eight reasoning failure scenarios depict how design reasoning can fail when vital information is not associated to key design elements.

Detailed discussions of our method is described in Section 3. We applied this method to review three industrial software architectures and we found 31 new design issues and 90 scenario instances (Section 4). We discuss the lessons learned from these case studies in Section 5, validity of our case studies in Section 6. Finally, Section 7 concludes the article.

2. Related work

Software design is a tricky business, especially designing highly complex software systems. Design is said to be wicked (Rittel and Webber, 1973), chaotic and ill-structured (Simon, 1973). The issue of designing software is not only about its design complexity, it is also about the ways people design. Humans are subject to cognitive biases during design (Stacy and MacMillan, 1995; Tang, 2011) and that can also cause design failures, resulting in poor quality software systems.

Many software architecture review techniques have been proposed in the research literature to validate architecture design and address the issues mentioned. On one hand, there are qualitative techniques such as Scenario-Based Analysis of Software Architecture (SAAM) (Kazman et al., 1994), Software Architecture Review and Assessment (SARA) (Obbink et al., 2002), Architecture Tradeoff Analysis Method (ATAM) (Bass et al., 2003), Lightweight Architecture Alternative Assessment Method (LAAAM) (Carriere, 2012) and Tiny Architectural Review Approach (TARA) (Woods, 2012). On the other hand, there are quantitative techniques such as Scenario-Based Architecture Reengineering (SBAR) (Bengtsson and Bosch, 1998) and Cost Benefit Analysis Method (CBAM) (Kazman et al., 2001). In this article, we propose an approach to qualitatively review an architecture design. We will give a brief overview of qualitative design review techniques and discuss their issues.

SAAM is a scenario-based review technique. The review of the architecture is based on the scenarios of the intended uses of the software. Since it is impractical, possibly infeasible, to generate all possible scenarios, architecture reviewers evaluate an architecture based on certain pre-selected scenarios. These scenarios are considered to be crucial to the quality of the architecture. As a result, the quality of the review depends heavily on the selected scenarios (Dobrica and Niemelä, 2002). For example, if the selected scenarios could not be used to identify or reveal the flaws in some critical assumptions and weaknesses of the architecture, there are problems in the "good-looking" architecture. Moreover, if the selected scenarios are incomplete in the sense that they cannot coherently address a particular requirement or their relative priorities are unclear, the reviewers need to make a lot of assumptions that may lead to biased results. Last, but not least, the selection of the scenarios might be biased as well.

The SARA Report (Obbink et al., 2002) provides guidance on conducting architectural reviews. The report describes the review inputs and outcomes in a review process and it suggests a checklist of assessments. The report has highlighted the complexity of such reviews.

The ATAM method is a scenario-based approach, and it faces the same scenario-dependent issues as SAAM. ATAM's focus is different from SAAM. It aims at helping stakeholders understand the tradeoffs of architecture decisions with respect to the system's quality attribute requirements. LAAAM is a variation of ATAM in which reviewers assess architectural strategies or principles against scenarios to evaluate architectural decisions (Carriere, 2012). This method does not specifically address the often complex interrelated design issues. TARA is a less formal approach than SAAM or ATAM. It is a lightweight architectural evaluation approach in which the reviewers explore and evaluate the context, requirements, software implementation and deployment of a system (Woods, 2012). This approach relies heavily on expert judgments and revealing deep architectural issues depends on their expertise.

Zannier et al. (2007) have found that the structure of problem space is important to design. The more structured the problem space is, the more rational the approach is taken by designers. Some designers use problem framing as a strategy to plan design activities and to manage complexity and the interplay of design components. Hall et al. (2002) and Nuseibeh (2001) suggest that problem frames provide a means of analyzing and decomposing problems, enabling a designer to iterate between the problem structure and the solution structure. Maher et al. (1996) suggest that solutions are developed as problems are identified during design explorations. These research have suggested that structuring problem space is an important element of design. However, the tracing of the problems and validating the proposed solutions against the identified design problems have not been a key element of any of the software architecture review methods.

Reviewers typically review architecture solutions, and not how the solutions are derived. This can be problematic as some designers do not explicitly identify the problems that they need to solve. There are implications in such situations. Firstly, design problems may be over simplified and not explored thoroughly. Secondly, a designer may anchor on a solution, and is reluctant to shift even in view of contradictory information. This has been found in the behavior of professional designers (Tang et al., 2010). Thirdly, designers may react to design issues instead of tracing the issues to their root causes. The synthesis of the problem and solution spaces requires software design knowledge as well as domain knowledge, that requires reflection-in-action (Schön, 1983). Design is also a critical conversation between ideation and evaluation (McCall, 2010). Reflecting on what problems a design must solve is therefore essential, especially when a reviewer is faced with a new and complex situation.

3. Architecture design decisions review framework

Large software systems require many architectural design decisions to be made, and many stakeholders make them. Some decisions are made in the early stages of a project, some are made during design, and even in implementation. As decisions are made over time by many different stakeholders, the decision makers may not be aware that there might be issues in them. These issues include incomplete information, ambiguous problem definition and so on. They can affect the quality of an architecture.

The methods software architects commonly used to record and communicate a design are text documents with design diagrams such as UML diagrams. Some organizations use CASE tools and traceability tools. However, design knowledge is often not sufficiently documented and communicated, they do not contain design reasoning. Many researchers have argued that designers are rationally bounded (Simon, 1996) and instead of rationalizing with a design, designers behave in an opportunistic way when designing (Guindon, 1990). One of the reasons is due to the limited cognitive capacity of designers and reviewers to process all requirements and design information simultaneously.

On this premise, designers can miss essential knowledge needed to design. If this essential knowledge can be readily accessed, such issues may potentially be avoided. In this proposed framework, we view a design as the result of many interrelated rational design decisions. Each design artifact is justified by the useful purposes it serves, such as some requirements. In this section, we describe design causal relationships and the different ways in which missing design knowledge negatively impacts on a design.

3.1. Design association

Design is a creative process (Simon, 1996). It is the ability of a designer to associate all relevant information and knowledge that they can access. In **Design Association Theory** (DAT), we theorize that architecture design is about **associating** key design elements and decisions, and reasoning with them to create a design.

An association is the establishment of causal relationships between key design elements. It has been suggested that design rationale should be created as a first class entity (Tyree and Akerman, 2005). Frameworks and standards are invented to depict the elements and the relationships in a design model. The main elements are design concerns (ISO/IEC, 2010), design decisions (Zimmermann, 2012; Jansen and Bosch, 2005) and design outcomes (ISO/IEC, 2010; de Boer et al., 2007). Some models such as the one suggested by Kruchten depict associations in terms of an ontology for design decisions (Kruchten, 2004). He suggests that decisions relate to each other in many ways such as constraining, subsuming and overriding. On the other hand, there are some general models that leave out specific relationship details. They model the causal relationships between design concerns and outcomes. One such model is Tang et al. (2006). We choose to use causal relationships to represent the causes and effects of a design because of its logicality and simplicity. For instance, a requirement motivates some design problems to be solved, and thus we end up with a software package. We suggest that such associations allow designers and reviewers to reason logically.

There are three design elements represented in DAT: (a) system concerns; (b) design problems; (c) design solution options. These are based on the ISO/IEC-42010 architecture rationale model shown in Fig. 1.

System concerns are the things that specify the goals of a design and influence design decisions. Examples are functional requirements, non-functional requirements, project contexts and technical constraints from a chosen operating platform. An *Architecture Decision* is about solving some kind of *design problem*. Design is a process of co-evolving between creating solutions and defining design problems (Dorst and Cross, 2001). As such, it is important to explicitly represent *design problem* in order for a solution to make sense. The outcome of a design is a *design solution* and some solution alternatives or options.

A typical architecture document describes the outcome of a design, using text, UML or other description languages. These design descriptions typically do not describe how a design is created, nor do they describe the implications of a design change. Little design rationale is provided. There is typically little information on whether all requirements are satisfied without any contradictions. Design Association Theory is aimed at overcoming this issue.

Although associations between key design elements are used, we differentiate between DAT and the idea of traceability such as Spanoudakis et al. (2004), Pinheiro (2000), and Ramesh and Jarke (2001). The former concept is aimed at discovering ideas and associating them together to reason with a design. The latter is aimed at creating and using trace links to help retrieve requirements and design information. The use of associations allows designers and reviewers to reason with a design, and additionally, it enhances the traceability of a design.

3.2. Graphical representation of design association

In order to apply DAT, we use the Architecture Rationale and Elements Linkage (AREL) meta-model to represent design decisions graphically (Tang and van Vliet, 2009), then we verify the architecture design by checking the associations and the design elements. Architecture design has many cross-cutting concerns, some of them can be easily omitted if an architect does not associate these concerns. To find any issues in an architecture design, architects and

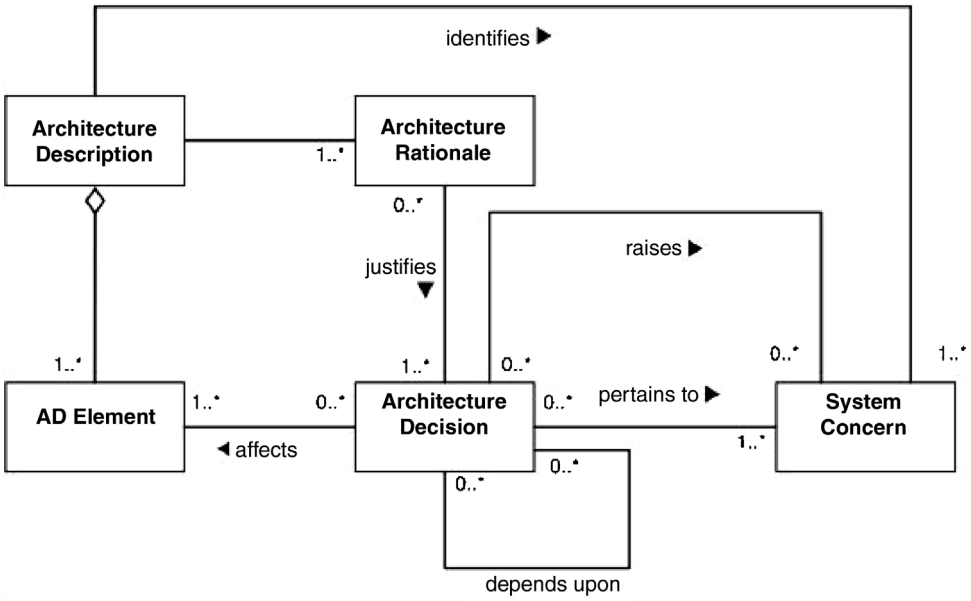

Fig. 1. ISO/IEC 42010 – architecture rationale.

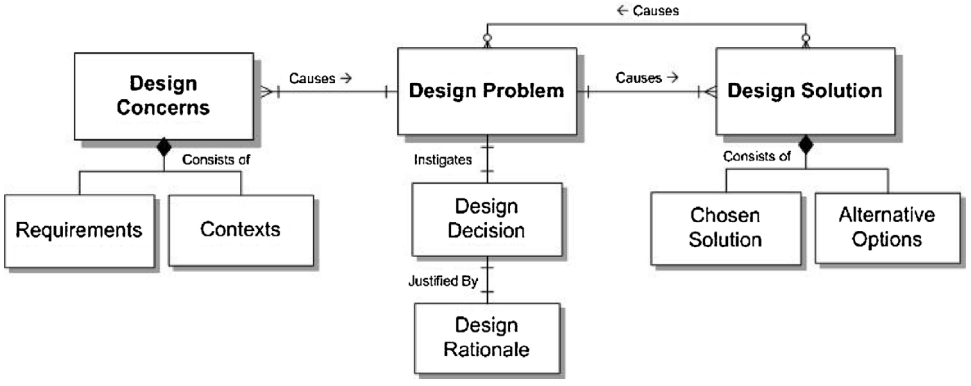

Fig. 2. A causal relationship of architecture decision making.

reviewers[1] need to understand design problems, realize what and how requirements can be satisfied, and assess the suitability of a design solution in achieving related goals and requirements. The AREL meta-model shown in Fig. 2 depicts a causal relationship. Its implementation is by using stereotyped classes in a UML tool called Enterprise Architect (EA).[2]

Design concern comprises design requirements and design contexts. By requirements we refer to architecturally significant requirements (ASRs) that have a major impact on the design of a software system (Bass et al., 2003). Both functional and quality attribute requirements are ASRs. *Design contexts* are conditions that influence design decisions but they are not specified explicitly as requirements. For instance, prior design choices, existing IT policy, skill-sets of the workforce, time-to-market needs and limited budget often create constraints and assumptions that influence design decisions.

Design problems arise from addressing design concerns. As software architecture design is about creating software artifacts to perform some actions, a design problem is about how to create those artifacts and define the behaviors of those artifacts. Postulating a design problem is an important step to exploring the solution (Maher et al., 1996). Explicit representation of the problem space is also an important step (Goldschmidt, 1997). For instance, a common design task of an architect is how to allow users to input some data. During the development of this solution, an architect needs to ask how to create a user interface that allows efficient data input, or reduce eye movements across the screen, or how to make this software flexible for data inputs in different business areas.

3.3. Design review procedure based on association

One objective of this research is to find a review procedure to help identify design issues based on causal relationships. To do so, we need to build causal relationships from existing system specifications. DAT underpins this model with which we construct a causal relationship model. We followed an iterative five-step review procedure: (1) extraction of requirements; (2) extraction of design; (3) construction of causal relationships; (4) discovery of potential design issues; (5) verification and confirmation of design issues. We suggest that parts of this procedure can be used during architectural design phase as well as for preparing a design.

[1] Architects can use such associations in their own design and decision making processes whereas reviewers can use the associations to evaluate the proposed design solutions.

[2] EA is available from Sparx and the AREL stereotype package can be found in http://www.ict.swin.edu.au/personal/atang/ArelStereotypePackage.zip.

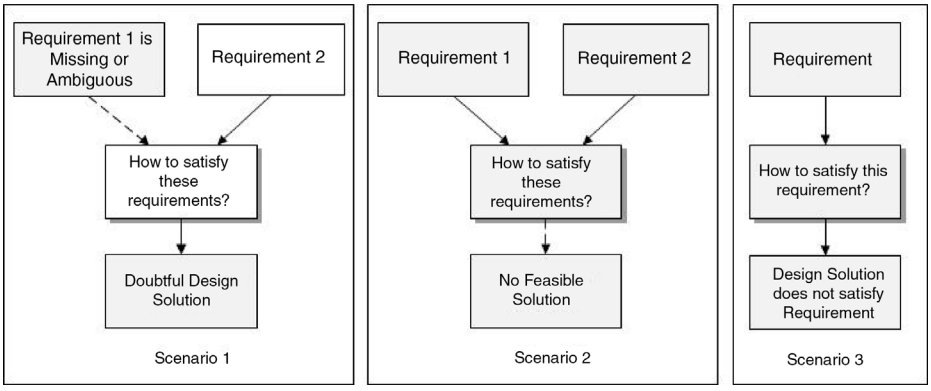

Fig. 3. Design reasoning failure – scenarios 1–3.

1. Extraction of requirements – The process of transforming from textual specifications to requirement nodes in a causal relationship took place semi-automatically. For requirements specifications that are well-structured and requirements systematically labeled, a tool was created to scan the specifications, in Microsoft Word format, and to extract the requirements automatically. The extracted requirements are created as *Design Concerns* nodes in a UML tool called Enterprise Architect (EA). Some requirement specifications are not systematically structured. If a requirement cannot be automatically identified by the scanning tool because they do not have a clear requirement label in the specification, they are manually added to the AREL model in EA. The requirements are manually grouped by functional requirements (FR) and non-functional requirements (NFR) under separate folders with the UML model, and then they are further manually sub-grouped by sub-systems or different types of NFRs. The grouping helps reviewers organize and find information.
2. Extraction of design – an extraction process is manually applied to the architecture design specifications, to transform textual specifications to design nodes in a causal relationship. Solution and solution alternative nodes are created in AREL models. The design nodes are grouped in accordance to the logical design of the system, such as by sub-systems, layers or components.
3. Construction of causal relationships – when the requirements and design models are in place, the associations between the causes and effects are reconstructed manually. The reviewers go through the graphs and reason how the requirements are causally related to the design. The design problems are constructed according to the requirements and contexts. The requirements and contexts are associated with design problems. If reviewers suspect that certain requirements or contexts might be missing, they can check with the architects. Reviewers associate each design problem to its design solutions. It is a process of visiting each node in a graph and asking ourselves some simple questions:
   - Design concern – what design problem this design concern might imply?
   - Design problem – what potential requirements or contexts might affect a design problem? What possible solutions are available?
   - Design solution – what problem does this solution solve? What requirements do this solution satisfy? What design problems matter to this solution?
4. Discovery of potential design issues – the basic premise of Design Association Theory is that a design solution is a logical consequence of the goals and problems that it is trying to solve. An AREL graph is constructed and analyzed to find relationships that do not make sense. At this stage, common design issues such as missing requirements and conflicting requirements are example issues to be mindful of. Any design concerns and design solutions, in the form of notes, that are considered illogical or doubtful are tagged. For instance, a solution that does not serve any particular requirement or a design problem is tagged with notes to indicate ambiguity (see the notes on the

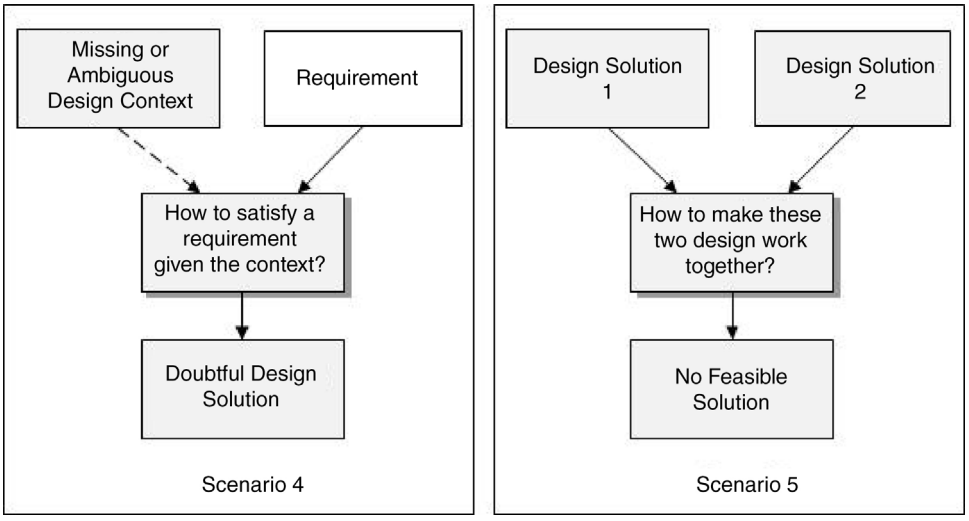

Fig. 4. Design reasoning failure – scenarios 4 and 5.

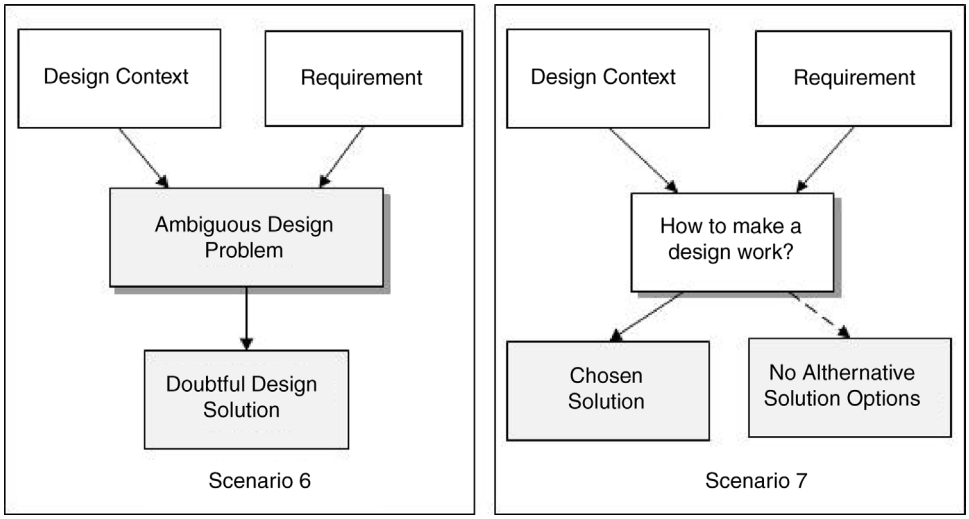

Fig. 5. Design reasoning failure – scenarios 6 and 7.

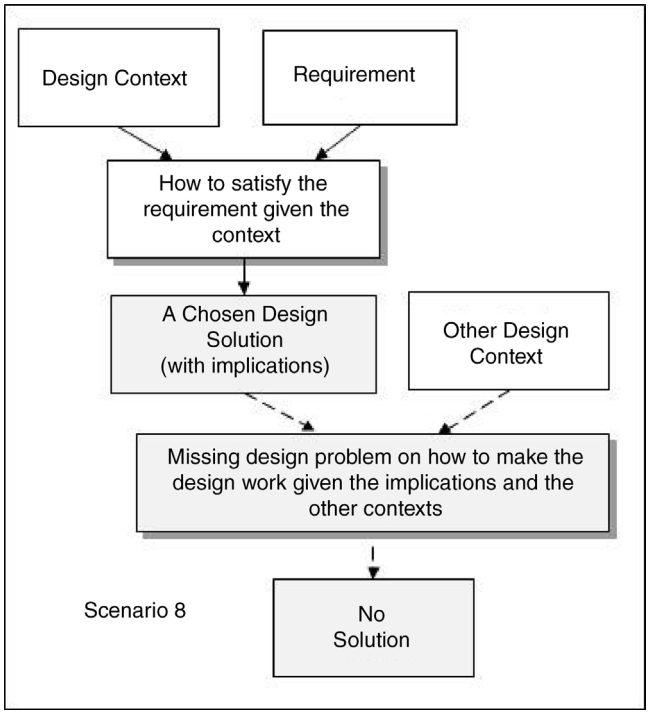

Fig. 6. Design reasoning failure – scenario 8.

right-hand-side in Fig. 7). Through this step, the associations and the design elements in the graph are questioned. It is possible to not find a logical answer using the documented elements in the graph. Such situations raise suspicions that something might be missing. The missing element might be a requirement, context, a problem or a solution option. These situations are flagged as potential issues for verification, and they can be checked with the architects through regular interactions such as conversations and meetings.

5. Verification and confirmation of design issues with architects – with a list of the suspected design issues, reviewers can discuss with software architects and designers to verify if these issues are real or not. The architects may confirm or reject the issues with explanations. In case of a confirmed issue, an architect can help to identify the root causes that lead to the issue, and explain the missing associations, or clarify the design considerations. Some issues that we have found in our study are rejected, they are issues that might be due to information missing from the specifications or it might be that reviewers misread the situation. Missing information maybe due to undocumented knowledge or misunderstanding. Such tacit knowledge can be stored as email, contained in someone's heads, and meeting minutes etc. In case of a confirmed issue, an architect can help to identify the root causes that lead to the design issue, and identify the missing associations and/or missing elements in the logical reasoning.

3.4. Common design reasoning failure scenarios

Meyer (1985) observed that common mistakes are made in design documentation. Design can have contradictions; requirements can be ambiguous; there are noises in design description, some aspects of a design can be omitted. Standard architectural description languages such as UML cannot fix these mistakes. Some researchers have investigated using formalism to help design. Formal representation of architecture model has limited success so far because it is hard to use. It is also difficult, if not impossible, to represent certain requirements and contexts (e.g., representing usability requirements formally). Informal architectural description such as text and diagrams are insufficient and imprecise, and formal architectural description cannot represent all architectural significant requirements. So what representation should be used by architects?

Architects and reviewers need to explore how design issues exhibit themselves in terms of missing design concerns, design problems, design solutions or the links between them. Our premise is that a coherent and logical design can be explained by its key design elements. So if some important premises or problem definitions are missing, architectural design issues may arise. Potential design issues could be identified through the lack of logical explanations of a design. For instance, the behavior of a solution does not satisfy any requirements or solve any problems. The identification of design issues is possible by viewing design concerns, design problems and design solutions together as causal relationships.

We anticipate some common design reasoning failures that cause design issues. We call them **design reasoning failure scenarios**. If architects and reviewers could easily recognize these failure scenarios, it might help them to identify design issues more easily. Below we outline eight design reasoning failure scenarios that are found in this study. The first five scenarios are easily

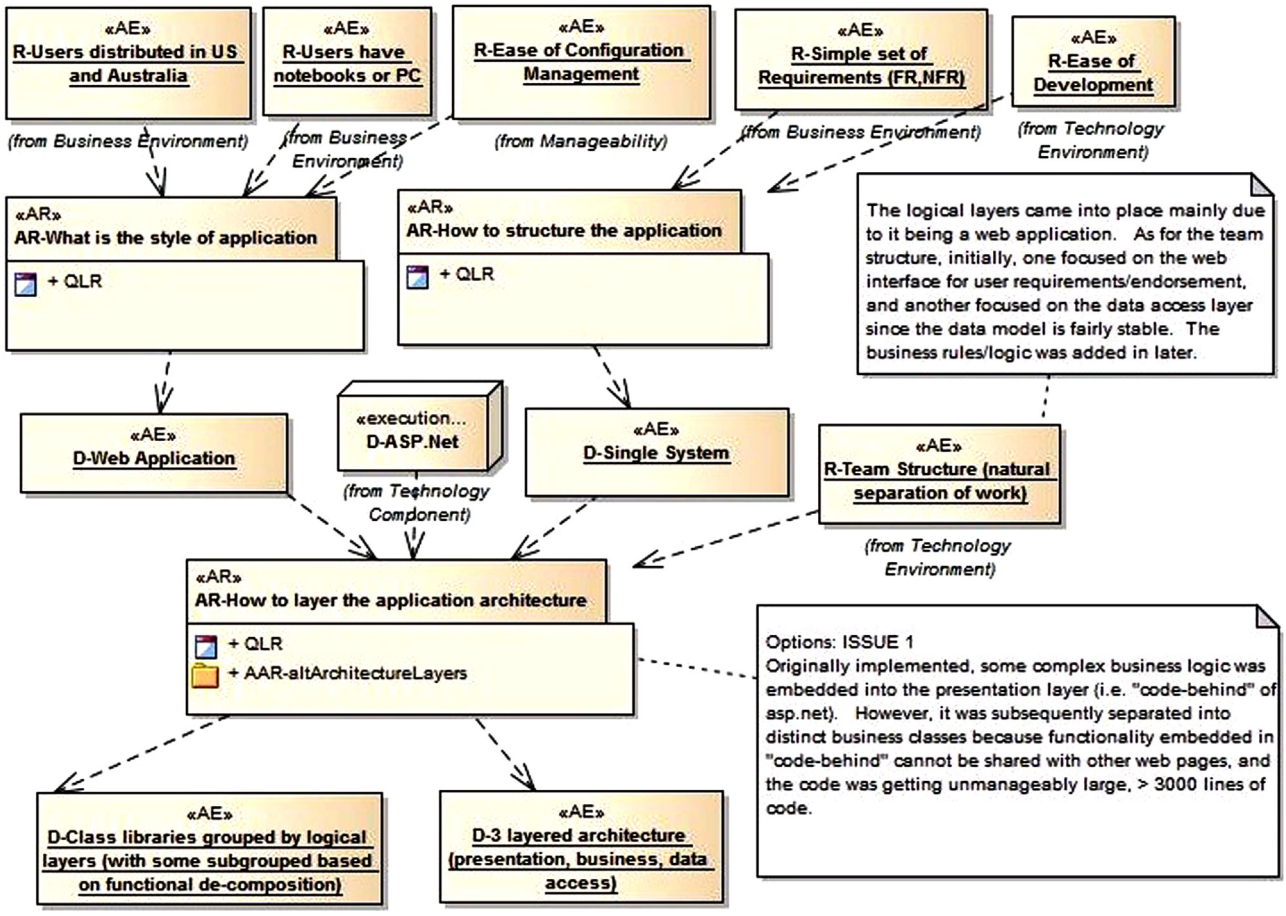

Fig. 7. Architecture verification example.

recognized. Actual design issues are found in all eight reasoning failure scenarios in our case studies.

1 **DI1-ReqMissing** *Requirement is missing or ambiguous* – when a requirement is missing in a design problem (meaning that it has not been specified), the architects and designers would have no knowledge about its existence and, hence, could not make the right design decisions. On the other hand, when a requirement is ambiguously defined in a design problem so that the architects and designers choose not to consider it when making the design decision (meaning that there is a missing association, denoted by dotted line/arrow, between the requirement and the design decision), the architects and designers again could not make the right design decisions. Another possibility of ambiguous requirement is that the requirement is not specified with regards to the design problem definition. Hence, certain details of a requirement are implicitly known to some architects and designers only. Such implicit details may contradict to or conflict with other design problem, requirement, context and solution. This situation could not create the right design decisions.

   In summary, when a requirement is ambiguous or missing in the design problem, the corresponding design solution is either incomplete or inaccurate. Scenario 1 in Fig. 3 depicts this situation.[3] A design solution was proposed by architects to satisfy known requirement (e.g., Requirement 2 in the figure). This solution may not work with missing or ambiguous requirements, and so its suitability becomes doubtful.

2 **DI2-ReqConflict** *Requirements conflicting each other* – when no possible solution is available to satisfy two requirements simultaneously, then we say that these two requirements are in conflict with each other. This situation happens when architects and designers never think to consider these two requirements together, or they do not frame the design problem correctly to bring them together. It may also be that requirements were proposed by different stakeholders, and if no one associates them as a single design problem, the conflict goes unnoticed. Scenario 2 in Fig. 3 depicts this situation. Reviewers using AREL could pick this up, whenever there is a need to simultaneously satisfy two requirements and there is no such design solution documented in the design. A more generalized situation would be multiple requirements conflicting with each other.

3 **DI3-ReqDesignConflict** *Requirements conflicting with design* – when a chosen solution does not satisfy a requirement. Assuming that a requirement is well specified, this can happen if a solution is created without considerations of the requirement it tries to fulfill. Hence, there is no justification of the chosen solution. Scenario 3 in Fig. 3 depicts this situation. Once the reviewers establish a causal relationship between a chosen solution and a requirement using AREL, they would then check whether the chosen solution could satisfy the requirement it needs to satisfy. If it could not, it falls into this design issue. A more generalized situation would be a chosen solution not being able to satisfy multiple requirements.

4 **DI4-ContextMissing** *Design context is missing or ambiguous* – design context describes factors that can shape a design

[3] In these AREL graphs, the shaded boxes depict problematic areas. Dotted lines depict missing associations.

problem and a solution. A context is an environmental factor that influences a decision. Contexts are not specified as requirements. Examples of contexts are: a technical constraint from a previously chosen design; a scheduled deadline; technological knowledge of personnel. They can influence a decision and make a large impact. Similar to the situation of missing requirements above, if a relevant context is missing resulting in making an implicit assumption, an architect may not have considered the context in their design problem definition. The resulting solution may not satisfy the needs. Scenario 4 in Fig. 4 depicts this situation.

Similar to the situation of ambiguous requirements, when a design context is ambiguous so that an architect chooses not to consider it when making the design decision (meaning that there is a missing association between the context and the decision). The resulting solution may not meet the demands. A more generalized situation would be multiple design contexts are missing or ambiguous.

5 **DI5-SolnConflict** *Design solutions conflicting each other* – two design solutions can conflict with each other. This can happen, whenever two design solutions are applied together, their behavior are inconsistent with each other or some functionality would not work properly, and there is no feasible solution to make them co-exist. Scenario 5 in Fig. 4 depicts this situation. A more generalized situation would be multiple design solutions conflicting with each other.

6 **DI6-ProblemMissing** *Design problem is missing or ambiguous* – a design problem is missing, ambiguous, ill-defined, undefined or irrelevant. Most of the time, reviewer has a solution but needs to associate the solution with the problem it tries to satisfy. When they tried to associate a solution to some design requirements or contexts, they had to reason what problems the architects were trying to solve. This issue occurs whenever a solution does not address relevant requirements and contexts, or it was unclear why the solution should exist. It could be because an architect does not clearly understand what design problem to solve. This is sometimes called a solution begging for a problem (see scenario 6, Fig. 5).

7 **DI7-NoAlternative** *No alternative design option* – this happens when one and only one single design option is considered where multiple and equally feasible options exist. Typically an architect does not justify the reason for choosing a solution. Alternatively, reviewers could find this out by asking why a particular solution was chosen where there could have been alternative(s); whether the architects considered alternative options; or whether they discarded any alternatives for real good reasons. The lack of considerations for other potential options limit the opportunity to find a more suitable solution, and to explore other related design problems, requirements and contexts that exist. This may be a result of an anchoring bias (see scenario 7, Fig. 5).

An architect may have considered the alternatives but not document them. A reviewer may then raise this as a design issue because as there is no way to tell whether design alternatives have been considered or not. If architects can demonstrate their considerations and alternatives, it is not an issue. Otherwise it is a potential issue.

8 **DI8-IgnoreConseq** *No consideration of further potential negative consequences* – when a solution is chosen, the impact of this solution on the rest of the design is not well considered. A chosen solution adds new contexts that influence further design, and some of these contexts may have negative impacts. Architects must consider what these implications are and what design problems might arise. If architects do not look-ahead, potential design problems can be ignored initially to surface later. For instance, choosing an open source software may have implications on software maintainability (see scenario 8, Fig. 6).

4. Industry case studies

Causal association between design elements such as requirements, design context, design problems and solutions underpins the Design Association Theory. In order to test the theory and the procedure, we chose to use a multiple-case study research methodology. Case studies require collecting empirical data in a real-life context for investigating some phenomenon (Yin, 2003; Verner et al., 2009).

We carried out three industry case studies. In each case study, the specifications were provided by the participating company, they were the up-to-date specifications of the systems. Three companies agreed to participate in this research. We obfuscate the information to assure confidentiality.

Case-Manufacturer was a system designed to monitor vehicle fleets. The system gathered data such as GPS location and vehicle information from Electronic Control Units (ECUs) in a fleet. Users interacted with the vehicles remotely through some communication networks. When we studied the case, this system was undergoing user acceptance testing phase.

Case-Engineering was a knowledge management system that supports collection and dissemination of engineering documents. Workflow processes were designed to manage document approvals and access. The requirement specification had been signed off but the design specification was yet to be reviewed.

Case-Travel was a loyalty system to monitor customers' activities and to support customers claiming rewards. When we studied the case, the system was already in production.

4.1. Case study method

In each case study, we spent approximately six man weeks in total. We used the five-step procedure defined in Section 3.3. We spent approximately four man weeks to extract the design concerns and the design solutions from the documented specifications in each case study (step 1 and step 2). A lot of time was spent on reading the specifications to ensure the correctness of data elicitation. The specifications were transformed into AREL models and causal relationships were constructed (step 3). When we had questions, we either phoned or met the architects to clarify the issues.

During model construction, we analyzed the model to discover potential issues (step 4). The identification of potential issues was continuous during the study. The verification of the results with the architects were through face-to-face meetings and telephone conferences. The AREL graphs were shown to the architects to explain the relationships we found during the study period and at the end of the study period. Any issues discovered were validated by the architects (step 5). At the end of each case study, we conducted a final meeting to summarize our findings and had the architects validate all of our findings.

Fig. 7 shows a typical example of an AREL graph that was constructed. The top left side of the graph shows the association between users' geography and the types of equipment they use and the architecture style (i.e. the choice of using web-based solution). The notes (partial) on the right hand side are annotations of a potential design issue to be verified.

After the design issues were confirmed with the architects, we analyzed the reasoning failures behind each identified issue. This is a step to generalize the issues into design reasoning failure scenarios by how a design issue comes to exist. The objective was to understand how the causal relationships in design broke down. Were some requirements missing? Or were some associations that should have existed between requirements and design problem went missing? From the analysis of the design issues, we have identified a total of eight general design reasoning failure scenarios. These design issues are instances of failure scenarios, we also call

them scenario instances. The process of identifying the scenario instances are as follows:

1. Examine a confirmed design issue to determine what was missing in the reasoning, in terms of the causal relationship.
2. Analyze the missing associations and information for mapping design issue to one or more design reasoning failure scenarios (see Table 2). For each issue, we often map to more than one design reasoning failure scenarios. For instance, a missing context is often coupled with ambiguous design problem.
3. In the cases of DI6-ProblemMissing, DI7-NoAlternative, and DI8-Ignore-Conseq, we started to see instances of these scenarios, and that allowed us to deduce that they are general scenarios.

Through this analysis, we confirmed the first five general design reasoning failure scenarios, and we discovered another three new design reasoning failure scenarios, DI6-ProblemMissing, DI7-NoAlternative, and DI8-IgnoreConseq, discussed in Section 3.4.

4.2. Research findings

With the graphs that were constructed, architects were asked to confirm or refute those identified potential design issues. The discussion between the architects and us then focused on the interpretations of the causal relationships. The explanatory power in the causal relationships helped to review the design correctness. This step seemed to avoid discussions based on subjective opinions such as a particular design choice is better or worse than another design choice, and allowed the discussions to focus on why the design existed.

Table 1 shows the summary statistics of the three case studies. Design concerns are design requirements and contexts. Design problems are the nodes that we constructed based on the requirements, contexts and solutions. Design solutions are elicited from the specifications. Documented alternatives are the alternative solutions that have been considered and documented in the specifications. Number of associations is a tally of the associations that we found. Potential issues are the issues that we discovered. If potential issues were confirmed by the architects, they become confirmed design issues. The AREL graphs were updated as we verified the design issues with the architects.

4.2.1. Instances of design reasoning failure scenarios

After confirming the design issues with the architects, we obtained a list of real design issues. We then analyzed the data to work out which design reasoning failure scenario an issue belongs to. This is a step to classify an issue by scenario to indicate what went wrong in terms of design reasoning.

Let us consider an example in Case-Engineering. One of the requirements was to let user access the data through the document management system (DMS) as well as using existing software such as email and through equipment such as mobile devices to get to that data. We questioned how the mobile devices could access DMS. After some exploration, it was found that current mobile devices are incompatible with the new DMS. Certain types of documents within the DMS such as emails and engineering drawings cannot be rendered to a mobile device. In this particular issue, we could see that it is a failure that can be explained by a number of scenarios:

1 The requirement is ambiguous (DI1-ReqMissing) – there is a blanket statement which states that all data in DMS can be accessed via mobile devices. There are no specifics to distinguish the different types of documents that need to be rendered to mobile devices.
2 Requirement conflicting with design (DI3-ReqDesignConflict) – when the requirement is clarified, it is found that the existing design cannot satisfy the requirements because some document types such as engineering drawings need specialized rendering software and cannot be shown on mobile devices.
3 No alternative solution (DI7-NoAlternative) – when deciding on which software to use to render to mobile devices, the chosen solution is the standard one offered by the DMS, there was no consideration of any other plugins that could be used to render the data.
4 No consideration of consequences (DI8-IgnoreConseq) – there is no consideration of how the DMS can integrate with the email system to extract the email item, or how special type of drawings such as AutoCAD can be rendered. It was also found that there was no consideration of the usability of the system on data conversion. These issues form a chain of questions that affect the viability of a solution.

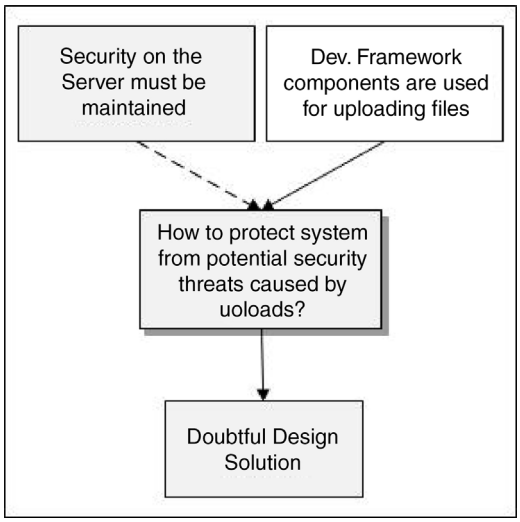

Fig. 8. An instance of ReqMissing in Case-Manufacturer.

When we analyzed each design issue using Design Association Theory, we observe that a design issue may have more than one design reasoning failure causes. Each of our proposed design reasoning failure scenario shows a cause of why a design issue happens. In the following, we selectively describe some of these failure scenario instances as examples.

Requirement is missing or ambiguous. This selected instance explains why a design solution is doubtful. In Case-Manufacturer, many files are required to be uploaded into the system. There is also a general requirement for security. When considering the requirement for uploading files, the architects did not consider the security requirement at the same time. The architects had not interpreted the general security requirement in terms of upload threat, and the architects did not specify, implicitly or explicitly, that there would be a design problem of tackling virus check in file uploads. As no virus checking requirement was specified, there is no design to cater for it. This is a case of a missing requirement. The highlighted areas in Fig. 8 show the issue.

Requirements conflicting each other. In Case-Engineering, one of the requirements for newly created documents in the knowledge management system (KMS) is to inherit document access control properties from the parent folder. However, there is a special type of document which access control properties cannot be inherited. Another requirement stated that the control properties of such special documents must be manually set-up and approved. These two requirements were in conflict with each other. This conflict was not noticed before our review. As a result of this discovery, the first requirement was modified to exclude the inheritance of the special

Table 1
Summary statistics (in total number) from the three case studies.

	Design concerns	Design problems	Design solns	Document'd alternatives	No. of Assoc'ns	Potential issues	Confirmed issues
Case-Manuf.	65	38	57	16	192	11	9
Case-Eng.	419	49	86	0	304	92	19[a]
Case-Travel	74	34	50	2	195	11	3

[a] We managed to confirm 19 design issues out of the 45 potential design issues, and the architects were not available to continue with the review dues to resignation. Otherwise, more design issues could possibly be found and our analysis could have been further improved.

document type. The conflict was resolved and a new solution was found (Fig. 9).

Requirements conflicting with design. In Case-Engineering, one requirement was to allow users to retrieve documents from a network drive. The proposed solution was to use the standard knowledge management system (KMS) user interface to access the network drive for the retrieval. This solution was infeasible. When we asked how documents could be transferred to the network drive, the architects then realized that the KMS package had no facilities to synchronize the files in KMS to the network drive in the current design. The requirement is in conflict with the design due to this newly discovered issue.

Design context is missing or ambiguous. One instance we found in Case-Manufacturer was the decision to use a dropdown list box that consisted of over 300 items prompting for user selection. In some applications such a design may be fine. For instance, when the items in the dropdown list box are ordered alphabetically. However, when this decision was made, the architect did not consider the limitation of the development framework that they had selected. The development framework provided a list box class that support displaying only 20 items at a time for user selection. Such a constraint in the development framework together with the design

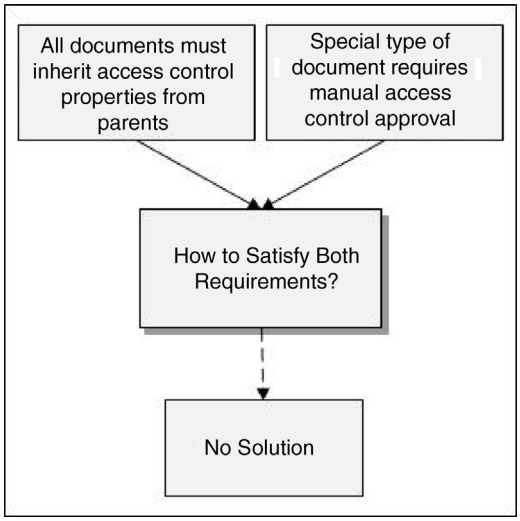

Fig. 9. An instance of ReqConflict in Case-Engineering.

Table 2
Design issues classified by scenario.

Issue Id	DI1	DI2	DI3	DI4	DI5	DI6	DI7	DI8
Case-Manufacturer								
Issue 1						X		
Issue 2						X	X	
Issue 3				X			X	
Issue 4							X	X
Issue 5	X			X				
Issue 6	X					X		X
Issue 7				X		X	X	X
Issue 8			X	X				
Issue 9				X		X	X	X
Case-Engineering								
Issue 1	X				X			X
Issue 2	X		X				X	X
Issue 3			X	X				
Issue 4	X		X	X		X		
Issue 5			X	X	X			
Issue 6		X		X				
Issue 7		X	X		X			
Issue 8		X	X	X	X			
Issue 9		X		X	X			
Issue 10	X	X		X	X			
Issue 11						X		X
Issue 12	X		X	X	X		X	X
Issue 13						X		X
Issue 14		X		X	X			
Issue 15	X	X	X		X			
Issue 16							X	
Issue 17	X		X	X		X		X
Issue 18						X	X	X
Issue 19				X		X		X
Case-Travel								
Issue 1			X	X				
Issue 2			X	X	X			
Issue 3						X		X

decision had created an usability issue for the users because users had to keep paging down 20 items at a time to the right selection. The presence or absence of the context (i.e. the display limitation of the list box) had made a difference to usability design. The selection of a solution would have been quite different had the designer realized the constraint of the development framework, i.e. taking this constraint as a context.

Design solutions conflicting each other. An example is from Case-Engineering. The knowledge management system (KMS) has to synchronize with Lightweight Directory Access Protocol (LDAP) periodically in order to update with the latest user security profiles. In order for this to happen, public access has to be enabled. However, enabling public access would mean that external users would have more privilege to access the KMS than should have been allowed during the synchronization process. The KMS and the synchronization process were in conflict with each other.

Design problem is missing or ambiguous. In Case-Travel, a requirement was for the website to achieve a certain performance level. There was a design to measure performance throughput. A packaged tool had been acquired for monitoring purpose. However, the problem of what exactly should be measured was undefined when the procurement decision was made. It turned out that the web service component needs the monitoring that this tool cannot measure. So the aim to monitor performance throughput could not be achieved because of an ill-defined problem.

No alternative design option. An example from Case-Engineering illustrates this design issue. In a business workflow, users required advance notification when certain action was due. When this solution was designed, the architects thought that this would work and did not consider another design option. It turned out that the product can support this but it required the users to have the expertise in setting up the facility manually. Not all users were able to do this. The alternative was considered after product release and it required to add new software. The new product was not in the original budget and therefore cannot be implemented. The users had to live with this shortcoming. In our case studies, we exclude those cases where the architects did consider alternatives but choose not to document them.

No consideration of further potential negative consequences. In Case-Manufacturer, an architect wanted to store a group of information in a database. For ease of retrieval, he decided to store into the database in XML format. He did not consider the consequence that in doing so, the data scheme had to be updated even with a minute change. When the schema needs to be maintained, the XML solution became unusable due to constant maintenance by a programmer. An alternative solution had to be found. In this case, a one step look-ahead is required to identify the design issue (see Fig. 10).

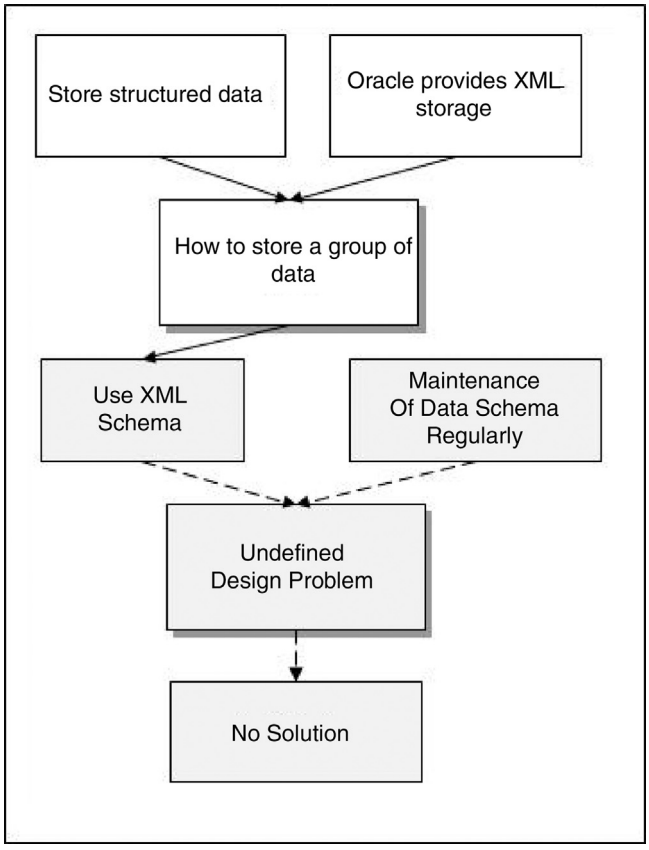

Fig. 10. An instance of IgnoreConsequence in Case-Manufacturer.

4.2.2. Scenario summary

We analyzed all the design issues in the three case studies to reason why reasoning failures occur. When a design issue occurs, there may be more than one reason why it happened. A summary of the scenario instances by failure scenarios is shown in Table 2.

We group the confirmed design issues into eight design reasoning failure scenarios. A tally of the reasoning failure scenario instances found in each case study is listed in Table 3. From the summary of design issues in Table 1 and the reasoning failure scenario instances in Table 3, we make a number of observations:

1 We note that architectural problems existed in all systems. Thus architectural reviews that the architects performed in these three systems do not guarantee to catch all design issues. We note further that the more mature a system is, the fewer design issues it contains. There are more confirmed design issues in Case-Engineering (total of 19) which was undergoing construction during our research period. In Case-Travel, which was a mature system that had undergone architectural reviews and had been in production for a few years, architectural design issues are fewer (a total of 3).
2 For DI1-ReqMissing and DI2-ReqConflict, there were more scenario instances in developing system (Case-Engineering and Case-Manufacturer) but no instances in the production system (Case-Travel). It is an indication that before a system is in place for production use, some reasoning issues that are caused by (1) ambiguous or missing requirements, and (2) conflicting requirements can be undetected.
3 For DI3-ReqDesignConflict, it is about requirements and design conflicting with each other. This scenario was subtle and, most of the time, was overlooked by designers and reviewers. However, when the system was in testing or production stages, and more information became available, there were fewer such failures.
4 For DI4-ContextMissing, it is about missing or ambiguous design context. There is a total of 18 scenario instances, which is approximately 20% of all scenario instances found. This is the most common scenario found in our studies. This design issues were subtle. It required the associating information from the context, problem and solution spaces to find them.
5 DI5-SolnConflict concerns about conflicting design solutions. It happened more often in new system than in mature system as the new system had not undergone testing.
6 DI6-ProblemMissing is about missing or ambiguous design problems. In our case studies, we identified many such instances in which the reason for a solution to exist was unclear. In other words, a solution is begging for a problem. This scenario was common to all three case studies. When we interviewed the architects to understand the contexts of their design, the architects also had difficulties tracking these design reasoning issues.

Table 3
Number of design reasoning failure scenario instances identified in the case studies.

Case/stage	DI1	DI2	DI3	DI4	DI5	DI6	DI7	DI8	Total
Case-Manuf./testing	2	0	1	5	0	5	5	4	22
Case-Eng./design	7	7	9	11	9	6	4	8	61
Case-Travel/production	0	0	2	2	1	1	0	1	7
Total	9	7	12	18	10	12	9	13	90

7 DI7-NoAlternative is about the lack of design options. This scenario was quite common in the design and testing stage. In our case studies, we could not find any such instance in the production system.
8 DI8-IgnoreConseq is about a shortfall of reasoning that ignores the consequences of a design decision. In such cases, this scenario is subtle, it happens frequently and it exists in all three case studies.
9 Among the 90 scenario instances, 56 of them belong to the first five scenarios, namely from DI1-ReqMissing to DI5-SolnConflict, and 34 belong to the last three scenarios, namely DI6-ProblemMissing, DI7-NoAlternative and DI8-IgnoreConseq. Design issues from this latter group are approximately one-third of the total instances found.

5. Discussions and insights

Traditionally, architectural design review relies on (1) the knowledge and experience of reviewers, (2) their ability to understand the application domain, the technical issues and the range of available solutions, and (3) their ability to judge if an architecture design is viable and can achieve the goals. Modern systems are large and complex, and hence, have many requirements, assumptions and constraints permeating throughout the systems, and many designers and stakeholders are involved. In such an environment, an ad-hoc human-based architectural review approach may not reveal deep design issues which are buried deep into a system.

In this study, we verified the software architecture designs of three real-life systems by checking the causal relationships of design reasoning. Before carrying out the verification, we were unfamiliar with the domains of the industry cases and so we cannot attribute our finding of design issues to our prior domain or software design experience. Through constructing the AREL models to represent the causal relationships of a design and systematic reasoning about the associations of various design elements, we have uncovered new design issues. Two of the three systems (Case-Manufacturer and Case-Travel) had already undergone architecture review process prior to our study, and yet we found new design issues via our association-based design review process. After the study, we learned that some of our findings helped the architects to improve their designs but we did not know how the architects dealt with the other findings.

The results of this study indicate that (a) not all design problems can be found after software testing or even when a system is in production; (b) conventional architectural review methods used by designers of these systems were insufficient to uncover all architectural design issues; and (c) almost 20% (18 scenario instances in DI4-ContextMissing out of all DIs) of the issues were caused by missing or ambiguous context. Therefore, tacit, ambiguous or missing context is one major cause of design reasoning issues.

Based on the results, we suggest that a systematic reasoning approach using associations can help to identify architectural design issues. In order to use this approach for software architecture review, there are four key considerations:

- Architectural knowledge can be tacit, and when they are not explicitly associated to relevant design problems, design gaps would occur. As such, explicit associations of design concerns, problems and solutions allow reviewers to identify issues that directly and indirectly influence decisions. This is especially useful in situations where (a) the design itself is complex and interim design decisions that are made by other designers are convoluted and tacit; (b) the contexts of a design decision is tacit.
- An explicit causal relationship helps to associate related concepts. Such association supports a creative process (Mednick, 1962) in which designers and reviewers would consider more associations.
- A *graphical* representation of hierarchical information has shown potentials to provide effective use of information because it helps users to understand complex structure relationships (Robertson et al., 1991) by reducing the cognitive load. A designer can see more of the concerns and problems at a glance. As one of the participating architects put it, "with this diagram, we can now explain to our business users why such a design decision has been made through a series of early design decisions".
- Eight general design reasoning failure scenarios, in a causal relationship form, have been identified. Five of the scenarios are known and three of the scenarios are new. They can be used as a checklist or reminders in design reviews.

A software architect who designs a new system needs all the relevant knowledge to make the right design decisions. This model helps designers and reviewers to associate architectural knowledge. However, if an architect lacks domain or technical knowledge, or does not communicate well with other stakeholders to get the relevant information, then failure scenarios like DI1-ReqMissing to DI4-ContextMissing would still occur. If an architect is knowledgeable, this method would provide a systematic method to help check for reasoning issues when dealing with complex and unfamiliar problems. This method would be less useful if a designer is already familiar with a design domain (Tang and van Vliet, 2012). Such a situation may be a bit too ideal. Nonetheless, our case studies demonstrate that our association-based approach helps design reviewers with less experience and domain knowledge than the architects identify important and new design issues.

The examples used in this article to illustrate design reasoning failures reflect complex real-life situations. We would like to further discuss the following points.

First, we explicitly define requirement (for the purpose of this study) as user needs that are documented in requirement specifications. On the other hand, a context can be many things including undocumented requirement, or conditions that simply influence decisions. A context is discovered when a designer or reviewer comes across an issue and recognizes that certain unsaid things are factors that influence a decision. During our discussions with the architects, some of these contexts were discovered, and some of them could have been documented requirements.

Second, on many occasions we questioned missing requirements and contexts. The knowledge was tacit and undocumented, but known by the architects. We did not record how many of these instances were found. However, the existence of these incidents demonstrated that when other stakeholders do not have the relevant tacit knowledge about a decision, they cannot trace or follow

the original logical design reasoning. As such, it may be useful to build the associations and causal relationships via AREL graphs during a design decision process to (a) capture important decision information whilst the tacit knowledge or the people who know that knowledge are still available; and (b) reduce the efforts of rediscovering the causal relationships by someone less intimate to the decisions.

Third, reasoning failure scenarios can be compounded. For instance, in DI8-IgnoreConseq, designers did not look ahead to consider the consequences of a design. This reasoning issue can also be attributed to missing context (DI4-ContextMissing) or to design question not been articulated (DI6-ProblemMissing). It may be difficult to argue what is the root cause of a design reasoning issue. Whilst such philosophical arguments might be intellectually intriguing to software design researchers, we believe that practitioners' focus is to recognize design reasoning failure scenarios so as to avoid design issues.

Fourth, DAT and its casual relationships are useful to validate a design. Through a structured reasoning approach, this method provides a means for architects to reason their design objectively and for reviewers to evaluate the chosen solutions objectively. The method may reduce decision biases. The creation of the casual relationship depends on many factors such as reviewer's experiences and their familiarity with the problem domain. This issue is faced by all architects and reviewers despite which review method they use. DAT has the advantage of depicting associations clearly so that it is easier for architects to trace ideas. This could help highlight missing design ideas. As shown in our case studies, our architecture reviewer(s) are considered to be less experienced and less familiar with the problem domain than the practicing architects, and yet with the causal relationship generated by DAT, we were able to identify new design issues that were overlooked by the architects. This demonstrates the usefulness of DAT and the causal relationships.

Fifth, even though DAT is proposed to support design review, it could also be used as a design technique in which architects reason their design and document the associations of their design elements. In fact, we would argue that, while designing, there are always reasons for architects to choose a design solution instead of another. Ideally, these reasons should all be captured and documented for review, and they should all be "objective" and "suitable for the purposes of the organizations".

Sixth, a key component of building causal relationships is the identification and articulation of design problems. This step depends largely on the knowledge and experience of an architect. Often this is performed implicitly by a software architect and they often do just enough to satisfice the goals. This way of design thinking can be explained by the opportunistic nature of an architect (Guindon and Curtis, 1988). As such, architects may not fully explore or articulate all the problems that need solving (Tang et al., 2010).

Seventh, this method requires architects to articulate design contexts, design problems and associations, on top of documented requirements and solutions. These are extra efforts to current design processes. However, the outcomes help designers and reviewers to analyze and review architecture design more effectively. The costs and benefits of such are good research for the future.

Finally, our proposed list of design reasoning failure scenarios is by no means exhaustive. We expect this topic to be investigated further and the list to be enhanced by us and others so that the entire software community will benefit from its existence. We also note that the procedures undertaken in our research is only the first step to demonstrate the use of such a method. Further work is required to refine the procedures for systematic design review.

6. Research validity

In this study, we used a multiple-case study to understand how reasoning failures might have caused software architecture design issues. We gathered design information to demonstrate that design issues can be revealed by inspecting design causal relationships. The construct of this study is underpinned by the Design Association Theory. The basic sources for architecture design validation are: (a) existing system specifications; (b) reconstruction of the causal relationships; and (c) verification of design reasoning by the architects. In all three case studies, the architects confirmed that the design issues that we identified, after eliminating false positives, were legitimate and they hope they could pick these issues up earlier.

Internal validity is concerned with the examination of the relationships between the research method (i.e. research steps) and the research data to ensure that there are no hidden variables that silently affect the investigated objects. In all three case studies, the method had shown to consistently helped to elicit the reasoning failure scenarios. The design issues found by the researchers were confirmed. Our research method was described to the architects as part of the research design.

External validity concerns the extent, to which the findings of the study are relevant for other situations and generalizable. As far as we can tell, there is no similar study on reasoning and software architecture design issues. However, results from cognitive psychology may explain why architects and reviewers had missed these issues. System complexity puts a high cognitive load on a designer, and missing knowledge creates reasoning gaps. Our method aims to detect reasoning failures and design shortfalls due to these reasons. However, this method relies on the knowledge and experience of the architects who employ it.

To provide independence of our research and analysis, we used an independent researcher (i.e. non-author) to interpret the specifications and to create the reasoning graphs. This independent researcher is a senior software architect who has been working in the IT industry for eighteen years. After the AREL graphs were created, all researchers checked the reasoning graphs before conducting interviews with the architects. The experience level of the researchers may have helped to identify some issues in these case studies. However, all the identified issues were based on the missing associations. Our findings have shown that it is feasible to logically identify design gaps through an association and reasoning process. When the architects walked through the associations, they checked and confirmed the design reasoning contained within.

7. Conclusion and future work

Software architecture reviews often depend on the experience and knowledge of the reviewers. As not all design knowledge is documented or shared, some knowledge that is essential to allow reviewers to identify design issues can be missing, and some knowledge that is available but not associated to the relevant design issues can also be missing. This can hamper the effectiveness of a design review.

We propose Design Association Theory in this paper. Design association is for associating design concerns, design problems and design solutions to show the causal relationships of why a design should exist. Associations also has the useful side effect of allowing designers and reviewers to trace design decisions. Through constructing such graphical models, we are able to discover new design issues in two systems that have gone through architecture review.

We propose an association-based design review process. It involves five steps, namely the extraction of requirements, extraction of design, construction of causal relationships between design

elements and design solutions, discovery of potential design issues, and verification and confirmation of design issues with architects. Our approach of using DAT is to help the reviewers to "re-capture" the design reasoning and decisions made by the architects while designing, and evaluate them objectively. In addition, architects could also use DAT to document their design decision, design reasoning to help evaluate their design objectively.

In this article, we performed and reported our findings of three industrial case studies, two of them had undergone internal architecture review. Our case study finds 90 design issues in these three real-life systems. Approximately one-third of these issues were found in the two systems that had gone through internal review. This shows that our technique is capable of finding a significant portion of new design issues that could not be found otherwise.

Using the discovered design issues of all three cases, we have identified and generalized eight design reasoning failure scenarios. Five of these eight scenarios are well known whereas the remaining three scenarios are found by reasoning about the causal relationships between design elements. These eight scenarios can help guide reviewers and designers to avoid reasoning failures.

Last, but not least, these eight design reasoning failure scenarios are by no means exhaustive. We would like to explore other design reasoning failure scenarios to help designers and reviewers. We also recognize that the procedures to elicit the design issues are simplistic. We would like to refine our procedures so that designers and reviewers can use them systematically for better reasoning.

References

Bass, L., Clements, P., Kazman, R., 2003. Software Architecture in Practice. Addison Wesley, Boston.

Bengtsson, P., Bosch, J., 1998. Scenario-based software architecture reengineering. In: Proceedings of Fifth International Conference on Software Reuse, pp. 308–317.

Brooks, F., 2010. The Design of Design: Essays from a Computer Scientist. Addison-Wesley Professional, Upper Saddle River, NJ.

Carriere, J., 2005. Lightweight Architecture Alternative Assessment Method (LAAAM). http://blogs.msdn.com/jeromyc/archive/2005/08/27/457081.aspx

de Boer, R.C., Farenhorst, R., Lago, P., van Vliet, H., Clerc, V., Jansen, A., 2007. Architectural knowledge: Getting to the core. In: 3rd International Conference on the Quality of Software-Architectures (QoSA), pp. 197–214.

Dobrica, L., Niemelä, E., 2002. A survey on software architecture analysis methods. IEEE Transactions on Software Engineering 28 (7), 638–653, http://dx.doi.org/10.1109/TSE.2002.1019479.

Dorst, K., Cross, N., 2001. Creativity in the design space: co-evolution of problem-solution. Design Studies 22 (5), 425–437.

Goldschmidt, G., 1997. Capturing indeterminism: representation in the design problem space. Design Studies 18 (4), 441–455.

Guindon, R., Curtis, B.,1988. Control of cognitive processes during software design: what tools are needed? In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. ACM, pp. 263–268.

Guindon, R., Krasner, H., Curtis, B., 1987. Breakdowns and Processes During the Early Activities of Software Design by Professionals. Ablex Publishing Corp., Norwood, NJ, pp. 65–82.

Guindon, R., 1990. Designing the design process: exploiting opportunistic thoughts. Human-Computer Interaction 5 (2), 305–344.

Hall, J., Jackson, M., Laney, R., Nuseibeh, B., Rapanotti, L., 2002. Relating software requirements and architectures using problem frames. In: IEEE Joint International Conference on Requirements Engineering, pp. 137–144.

ISO/IEC, 2010. ISO/IEC CD1 42010 Systems and software engineering ? Architecture description).

Jansen, A., Bosch, J., 2005. Software architecture as a set of architectural design decisions. In: Proceedings 5th IEEE/IFIP Working Conference on Software Architecture, pp. 109–120.

Kazman, R., Bass, L., Abowd, G., Webb, M., 1994. SAAM: a method for analyzing the properties of software architectures. In: 16th International Conference on Software Engineering, 1994, Proceedings. ICSE-16, pp. 81–90, http://dx.doi.org/10.1109/ICSE.1994.296768.

Kazman, R., Asundi, J., Klein, M., 2001. Quantifying the costs and benefits of architectural decisions. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001), pp. 297–306.

Kruchten, P., 2004. An ontology of architectural design decisions in software-intensive systems. In: 2nd Groningen Workshop on Software Variability Management.

Maher, M.L., Poon, J., Boulanger, S., 1996. Formalising design exploration as co-evolution: a combined gene approach. Tech. Rep. University of Sydney.

Maranzano, J.F., Rozsyal, S.A., Zimmerman, G.H., Warnken, G.W., Wirth, P.E., Weiss, D.M., 2005. Architecture reviews: practice and experience. IEEE Software 22 (2), 34–43, http://dx.doi.org/10.1109/MS.2005.28.

McCall, R., 2010. Critical conversations: feedback as a stimulus to a creativity in software design. Human Technology 6 (1), 11–37.

Mednick, S.A., 1962. The associative basis of the creative process. Psychological Review 69, 220–232.

Meyer, B., 1985. On formalism in specifications. IEEE Software 2 (1), 6–26.

Nuseibeh, B., 2001. Weaving together requirements and architecture. IEEE Computer 34 (3), 115–119.

Obbink, H., Kruchten, P., Kozaczynski, W., Postema, H., Ran, A., Dominick, L., et al., 2002. Software architecture review and assessment (SARA) report (version 1.0). Tech. Rep.

Parnas, D., Weiss, D., 1987. Active design reviews: principles and practices. Journal of Systems and Software 7 (4), 259–265, http://dx.doi.org/10.1016/0164-1212(87)90025-2 http://www.sciencedirect.com/science/article/pii/0164121287900252

Perry, D.E., Wolf, A.L., 1992. Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17 (4), 40–52.

Pinheiro, F.A.C., 2000. Formal and informal aspects of requirements tracing. In: Workshop em Engenharia de Requisitos. Brazil, pp. 1–21.

Ramesh, B., Jarke, M., 2001. Towards reference models for requirements traceability. IEEE Transactions on Software Engineering 27 (1), 58–93.

Rittel, H.W.J., Webber, M.M., 1973. Dilemmas in a general theory of planning. Policy Sciences 4 (2), 155–169.

Robertson, G.G., Mackinlay, J.D., Card, S.K.,1991. Cone trees: animated 3D visualizations of hierarchical information. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Reaching through Technology, CHI'91. ACM, New York, NY, USA, pp. 189–194, http://dx.doi.org/10.1145/108844.108883, ISBN 0-89791-383-3.

Schön, D.A., 1983. The Reflective Practitioner: How Professionals Think in Action. Basic Books, NY, USA.

Simon, H.A., 1973. The structure of ill structured problems. Artificial Intelligence 4 (3–4), 181–201, http://dx.doi.org/10.1016/0004-3702(73)90011-8.

Simon, H., 1996. The Sciences of the Artificial. MIT Press, Cambridge, Massachusetts; London, England.

Spanoudakis, G., Zisman, A., Pérez-Miñana, E., Krause, P., 2004. Rule-based generation of requirements traceability relations. Journal of Systems and Software 72 (2), 105–127.

Stacy, W., MacMillan, J., 1995. Cognitive bias in software engineering. Communications of the ACM 38 (6), 57–63.

Tang, A., van Vliet, H., 2012. Design strategy and software design effectiveness. IEEE Software 29 (1), 51–55, http://dx.doi.org/10.1109/MS.2011.130.

Tang, A., van Vliet, H., 2009. Software Architecture Design Reasoning. Springer, Berlin Heidelberg, pp. 155–174.

Tang, A., Jin, Y., Han, J., 2006. A rationale-based architecture model for design traceability and reasoning. Journal of Systems and Software 80 (6), 918–934.

Tang, A., Aleti, A., Burge, J., van Vliet, H., 2010. What makes software design effective? Design Studies 31 (6), 614–640, http://dx.doi.org/10.1016/j.destud.2010.09.004.

Tang, A.,2011. Software designers, are you biased? In: Proceeding of the 6th International Workshop on SHAring and Reusing Architectural Knowledge; SHARK'11. ACM, pp. 1–8, ISBN 978-1-4503-0596-9.

The Open Group, 2003. The Open Group Architecture Framework (v8.1 enterprise edition). The Open Group http://www.opengroup.org/architecture/togaf/#download

Tyree, J., Akerman, A., 2005. Architecture decisions: demystifying architecture. IEEE Software 22 (2), 19–27.

Verner, J., Sampson, J., Tosic, V., Bakar, N., Kitchenham, B., 2009. Guidelines for industrially-based multiple case studies in software engineering. In: Research Challenges in Information Science, 2009. RCIS 2009. Third International Conference on, pp. 313–324, http://dx.doi.org/10.1109/RCIS.2009.5089295.

Woods, E., 2012. Industrial architectural assessment using tara. Journal of Systems and Software 85 (9), 2034–2047.

Yin, R., 2003. Case Study Research Design and Methods. Applied Social Research Methods Series, 3rd ed. Sage Publications, London.

Zannier, C., Chiasson, M., Maurer, F., 2007. A model of design decision making based on empirical results of interviews with software designers. Information and Software Technology 49 (6), 637–653.

Zimmermann, O.,2012. Architectural decision identification in architectural patterns. In: Proceedings of the WICSA/ECSA 2012 Companion Volume, WICSA/ECSA'12. ACM, New York, NY, USA, pp. 96–103, http://dx.doi.org/10.1145/2361999.2362021, ISBN 978-1-4503-1568-5.

Antony Tang received the PhD degree in information technology from the Swinburne University of Technology. He is an associate professor in Swinburne University of Technology's Faculty of Information and Communication Technology. Prior to being a researcher, he had spent many years designing and developing software systems. His research interests include software architecture design reasoning, software development processes, software architecture and knowledge engineering. He is a member of the ACM and the IEEE.

Dr Man Fai Lau is a Senior Lecturer in the Faculty of Information and Communication Technologies, Swinburne University of Technology in Australia. He received his BSc(Hons) degree from the University of Hong Kong and PhD degree in Software Engineering from the University of Melbourne. His research interests include software testing, software quality, enterprise systems, software specification and computers in education.

To date, he has received approximately AUD$800K (equiv.) of competitive research grants including Australian Research Council Discovery Project Scheme (ARC DP) and Hong Kong Research Grants Council Competitive Earmarked Research Grant Scheme (HKRGC CERG). His research publications have appeared in many internationally reputable scholarly journals, including ACM Transactions on Software Engineering and Methodology, The Computer Journal, Information Processing Letters, Information Sciences, Information and Software Technology, The Journal of Systems and Software, and Software Testing, Verification and Reliability.

Dr. Lau has been served as an international reader (a reviewer of international standing) of the ARC grant proposals as well as HK RGC grant proposals. He has also served as a reviewer of many scholarly journals such as IEEE Transactions on Software Engineering, IEEE Software, Information Processing Letters, Information and Software Technology, Journal of Systems and Software, and Software Testing, Verification and Reliability; and serverd as program committee members of various international conferences and workshops.