11 A Framework for Supporting Architecture Knowledge and Rationale Management

M.A. Babar, I. Gorton, B. Kitchenham

Abstract: There is growing recognition of the importance of documenting and managing background knowledge about architecture design decisions. However, there is little guidance on the types of information that form Architecture Design Knowledge (ADK), how to make implicitly described ADK explicit, and how such knowledge can be documented to improve architecture processes. We propose a framework that provides a support mechanism to capture and manage ADK. We analyze different approaches to capturing tacit and implicit design knowledge describe a process of extracting ADK from patterns and an effective way of documenting it. We also present a data model to characterize architecture design primitives used or generated during architecture design and evaluation. This data model can be tailored to implement repositories of ADK. We complete this chapter with open issues that architecture research must confront in order to successfully transfer technology for capturing design rationale to the industry.

Keywords: software architecture; design rationale; knowledge management

11.1 Introduction

Many researchers and practitioners acknowledge the importance of capturing and maintaining knowledge underpinning architecture decisions [17, 42, 52]. However architecture/design decisions are seldom documented in a rigorous and consistent manner. Curtis et al. [19] and Bosch [12] suggest that a meaningful explanation should include information explaining the context, reasoning, tradeoffs, criteria, and decision making that led to the selection of a particular design from various design options. This type of knowledge is called *design rationale (DR)* [34, 43]. It represents knowledge that provides the answers to questions about a particular design choice or the process followed to make that choice [20, 24]. If it is not documented, knowledge concerning the domain analysis, patterns used, design options evaluated, and decisions made is lost, and so is unavailable to support subsequent decisions in development lifecycle [12, 41, 52].

Based on our experiences in designing and evaluating architectures for large-scale systems, we argue that lack of suitable techniques, tools, and guidance is one of the reasons that DR is not captured and managed. DR researchers have developed different methods, notations, and tools for recording design decisions, such as IBIS [32], Decision Representation

Language (DRL) [34], gIBIS [18], Questions, Options and Criteria (QOC) [35], and so on. However, these approaches only capture and represent the space or history of arguments surrounding the design decisions [15], rather than representing domain knowledge or system design in terms which are understandable by the domain experts [26]. Moreover, these approaches are not scaleable to large scale systems; nor do they ensure creation of reusable assets, promote the use of DR, or improve the reuse of artifacts [25].

We propose a framework for managing DR to improve the quality of architecture process and artifacts. This framework consists of techniques for capturing DR, an approach to distill and document architectural information from patterns, and a data model to characterize architectural constructs, their attributes and relationships. These collectively comprise Architecture Design Knowledge (ADK) to support architecting process. The central objective of our research is to develop a generic framework for capturing architecture process knowledge (DR attached to artifacts) and provide mechanism to manage the captured knowledge to support architecture design decision making process.

In this chapter, like [20], we characterize DR as certain type of architecture knowledge developed and used during software development. We take the view that management of such knowledge can be greatly improved by considering the various tasks from a management perspective rather than computer science or artificial intelligence perspective [31] . Thus, our approach considers DR management tasks as Knowledge Management (KM) tasks and uses a KM task model described in [46] and used in [20, 31]. This model consists of two strategic and six operational tasks.

11.2 Background and Motivation

11.2.1 Design Rationale Approaches and Software Engineering

A DR is an explanation of how and why an artifact, or some part of it, is designed the way it is. DR represents knowledge about the reasoning justifying the resulting design. This includes how a structure satisfies functional and quality requirements, why certain structures are selected over alternatives, and what type of behavior is expected under different environmental conditions [24, 34]. Early research emphasizing the importance of DR in software design can be found in [40, 43]. Since then, the software engineering community has experimented with several DR approaches such as Issue-Based Information Systems (IBIS) [32], Questions, Options, and Criteria (QOC) [35], Procedural Hierarchy of Issues (PHI) [36], and

Decision Representation Language (DRL) [34]. All these approaches provide argumentation models, which use small numbers of node and link types to organize a hierarchy of questions posed to address issues. Alternatives, their rationales, and the final choice can be attached to the questions to document discussion paths [25].

Most of these approaches have been adopted or modified to capture rationale for software design decisions [33, 43] and requirements specifications [21, 47, 51]. Another approach [45] combine rationale and scenarios during requirements elicitation process to refine and review requirements. Dutoit and Paech [20] describe various tasks of rationale management in software engineering and their application. Heninnger developed an approach for supporting reuse-based software development by explicitly capturing and using past rationale [25]. Pena-Mora and Vadhavkar's DRIM [41] approach combines patterns and rationale to support reusable software development. Our approach has some similarities with the last three approaches. But, instead of attaching DR to patterns like the DRIM, we distill architectural information from patterns and represent it as a reusable ADK. We believe that rationale should be attached to design decisions, which apply patterns, rather than patterns themselves.

Software architecture (SA) researchers have also emphasized the need to document DR to maintain and evolve architectural artifacts and to avoid violating fundamental rules underpinning the original design decisions [8, 12]. The IEEE 1471 standard [27] identifies DR as an important part of SA description. However, there has not been any significant support mechanism for capturing and managing rationale for architecture decisions except the *Views and Beyond* (V&B) approach to document SA [17]. Though, the V&B approach recommends explicit documentation of rationale for design decisions, interface designs and the information that cuts across multiple views, it also has certain limitations mentioned later.

11.2.2 Relationship among Quality Attributes, Scenarios, and Patterns as Architecture Design Knowledge

A quality attribute is a nonfunctional requirement (NFR) of a software system, such as maintainability, performance and so forth. Scenarios have been found effective and useful for specifying quality attributes. That is why scenarios are widely used to design and evaluate SA. Scenarios are considered flexible as they can be used for systematically reasoning about, or evaluating, most quality attributes [2, 29]. A pattern is a known solution to a recurring problem in a particular context. Patterns provide a mechanism for documenting design knowledge [22]. The architecture of complex

systems is usually designed by successively integrating different patterns, which may be described at different levels of abstraction. Each pattern supports or inhibits certain quality attributes [8, 14].

Patterns documentation also contains reasons for the use of a pattern for a certain class of problems. Relating quality attributes, scenarios, and patterns in this way forms architecturally significant knowledge, which may also have rationale about the relationships attached. Recently, there have been a few efforts to explicitly codify the relationships among quality attributes, scenarios, and patterns [9, 23]. These approaches identify and link scenarios, quality attributes, and patters from sources other than the patterns themselves. However, we have demonstrated that each pattern's documentation contains implicit description of the relationships among scenarios, quality attributes, and patterns [3].

Having reviewed various approaches to rationale management, we conclude that most of the efforts to introduce argumentation methods in software engineering have experienced very limited success. There are many reasons for this, for example, the need for extensive training, changes in thinking styles, focus on some tasks only, and lack of guidance on using past rationale [13, 25]. Generally, knowledge and rationale management support in the architecture domain is limited.

Obbink et al. [39] and the IEEE 1471 standard [27] advocate capturing and maintaining rationale but do not provide any support mechanism. V&B provides templates to capture knowledge about DR, which is documented along with the architecture description. However, there is no sufficient support mechanism to capture and manage knowledge and rationale about other architectural constructs such as scenarios, patterns and so on, and their relationships. Moreover, none of the existing approaches in the architecture domain help users to identify and define the main constructs and their properties and relationships involved in forming ADK. Nor do they provide sufficient conceptual guidance to develop a repository of ADK and experiences of using it [6].

The main object of this research is to develop a support mechanism for capturing and maintaining ADK to improve architecting process. To achieve this goal, the issues that we intend to address are to identify techniques that make the effort of capturing rationale worthwhile without heavily disrupting the design process, help utilize the large amount of architecturally significant information implicitly embedded in patterns, and to identify architectural constructs, their properties and relationship and put them into a framework that can be used to design and implement an organizational repository to store and retrieve ADK generated or used during architecting process. In Sect. 11.3, we propose a solution that

incorporates techniques that complement each other and provide an integrated support framework for capturing and maintaining ADK.

11.3 Managing Architecture Design Knowledge

In this section, we present a framework for managing ADK . This framework comprises three components:

1. A means of capturing knowledge underlying decisions from architects as well as electronic sources such as annotations attached to artifacts.
2. A procedure for capturing architecture knowledge and associated rationale from patterns in order to explicate the relationships among scenarios, quality attributes, and patterns that is a form of reusable ADK.
3. A model for characterizing the main architectural constructs and their relationships that form ADK and rationale. This data model can be tailored and implemented to provide a repository for managing process knowledge, relating design knowledge to architecture artifacts or the reusable ADK extracted from patterns.

These three components complement each other to support the tasks of capturing ADK from different sources (such as architects, artifacts, and patterns), structuring and maintaining the captured ADK, which is presented in a format that is readily usable in making and assessing design decisions with an informed knowledge of the consequences of those decisions. The first two components are aimed at capturing ADK, while the third component represents architecture domain knowledge that can help develop a knowledge base to store, maintain, and retrieve the captured ADK. The balance of this chapter is heavily skewed towards the *pattern-mining* (3.1) and the data model (3.2)) components of the framework. However, we discuss briefly three approaches that are being used to capture DR, process knowledge, or experiences in software engineering:

1. Designers themselves can be required to document their DRs [4]. Knowledge can be captured during the design process or constructed after the fact [43]. In either case, the designer needs to be motivated to document the DR. In practice such motivation depends on appropriate rewards and explicitly demonstrated future benefits [25].
2. A knowledge engineer [50] or rationale maintainer [20] can be appointed to the task of capturing design knowledge from designers, meeting recordings, emails, memos, and design documentation. Industrial trials conclude that a KM tool and a knowledge engineer should be an integral part of software development process [50]. However, this

approach should be used with caution as the knowledge engineers may become a bottleneck [25].

3. ADK can be captured during architecting process. This is called *contextualised* knowledge. Designers are provided with appropriate tools so that knowledge can be encoded into the system as part of the knowledge creation process [25]. This is similar to the V&B approach, which provides templates to capture contextualized design knowledge [16].

We find the second and third approaches less disruptive and useful. We are also studying the architectural processes to integrate design knowledge capture practices in a manner that is not overly disruptive [25].

11.3.1 Mining Patterns for Architecture Knowledge

The idea of mining patterns for architectural significant information was conceived as part of our efforts to improve SA evaluation. We found that software patterns are a valuable source of abstract scenarios, which can be distilled to support SA evaluation. We later found that each pattern's description is also a source of architecturally significant relationships that exist among scenarios, quality attributes, and patterns. We argued that such synergistic relationships form ADK that needs to be explicitly documented in a readily reusable format to support architecting process.

Our initial experiences of capturing architectural artifacts and relationships among them from patterns were encouraging. However, we found that being a manual procedure it relies heavily on the pattern miner's experience with different classes of patterns (such as architectural, design and platform specific) and with several formats of documenting patterns. In addition, the extracted information needs to be documented in a format that explicates the relationships among scenarios, quality attributes, and patterns as ADK along with the rationale for using a pattern. Thus, we have developed an approach to identify, capture and document architecturally significant information from patterns as ADK.

This approach consists of a process model, guidelines, and a template to identify, capture, and document architecturally significant information from patterns as architecturally significant reusable artifacts. We call this process "pattern-mining" and the extracted information "Architecturally Significant Information extracted from Patterns (ASIP)" [3]. The novelty of our approach resides in its ability to incorporate all the components into an integrated approach, which can be used to capture and populate ADK repository like [38] backed by an experience factory infrastructure [6] to grow organizational capabilities in architecting process.

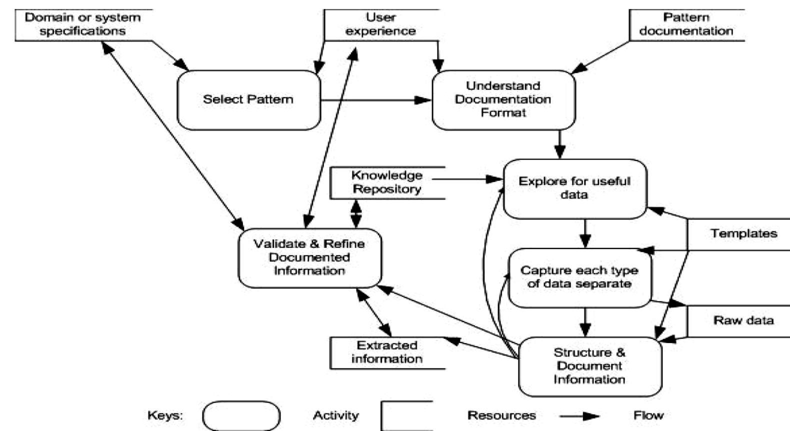

Fig. 11.1. Pattern-mining process model

The Pattern-Mining Process

The *pattern-mining* process is shown in Fig. 11.1. This manual process consists of following six steps:

1. Select a software pattern to be explored for architectural information
2. Understand the pattern documentation format to identify the variations that exist among different patterns' description styles
3. Explore different parts of the selected patterns to identify architectural information described in a pattern's documentation
4. Capture each type of information separately
5. Structure and document the extracted information
6. Validate and refine documented information based on domain knowledge and experience of using different patterns

Patterns are usually documented in a variation of format used in [22], which require the inclusion of problem, solution, and quality consequences parts. We have found that scenarios are mostly found in problem and solution sections. Forces can also be found in these sections. However, there are some pattern description styles that use separate sections for forces. The forces of a pattern describe the factors which can cause a problem if they interfere with one another. The pattern attempts to resolve clashes among those factors. Discussion of forces also captures necessary tradeoffs and justification for using that pattern, which is considered the DR of that pattern [23, 28]. The quality attributes (positively or negatively affected) are usually described at the end of a pattern's description.

Documenting and Representing Design Knowledge

The *ASIP* must also be documented and presented in a format, which turns it into architecture design knowledge that can facilitate reasoning during architecture process. We have designed and assessed a template (Table 11.1) to document and represent architectural constructs, including abstract scenarios, quality attributes, forces, tactics, and usage examples. This template makes the synergistic relationship among scenarios, quality attributes, and patterns explicit, which forms ADK. The template also presents different parts of a pattern's description in a succinct format at an abstraction level suitable for architecture design and evaluation. Since designers usually apply breadth first strategy to identify available solutions to a design problem [49], too much detail presented in the current formats of pattern documentation may be counter-productive [3].

Table 11.1 A template to document ADK extracted from patterns

<table>
<tr><td colspan="3">pattern name: name of the pattern</td><td colspan="2">pattern type: architecture, design, or style</td></tr>
<tr><td colspan="2">Description</td><td colspan="3">a brief description of the pattern</td></tr>
<tr><td colspan="2">Context</td><td colspan="3">the situation for which the pattern is recommended</td></tr>
<tr><td colspan="2">Problem</td><td colspan="3">what type of problem the pattern is supposed to address?</td></tr>
<tr><td colspan="2">suggested solution</td><td colspan="3">what is the solution suggested by the pattern to address the problem?</td></tr>
<tr><td colspan="2">Forces</td><td colspan="3">factors affecting the problem and solution and pattern's justification.</td></tr>
<tr><td colspan="2">Tactics</td><td colspan="3">what tactics are used by the pattern to implement the solution?</td></tr>
<tr><td colspan="2" rowspan="2">affected attributes</td><td colspan="2">positively</td><td>negatively</td></tr>
<tr><td colspan="2">attributes supported</td><td>attributes hindered</td></tr>
<tr><td rowspan="2">abstract scenarios</td><td>S</td><td colspan="3">a textual, system independent specification of a quality attribute.</td></tr>
<tr><td>S</td><td colspan="3"></td></tr>
<tr><td>example</td><td colspan="4">some known examples of the usage of the pattern to solve the problems</td></tr>
</table>

Apart from providing a structured way of documenting the *ASIP*, the template also support the *pattern-mining* process by helping a pattern-miner (e.g., software architect) concentrate on the pieces of information that need to be extracted to populate the template. Moreover, the ADK presented in this template enables a user of the template to reason about the ramifications of the tactics being implemented by a particular pattern within the context of scenarios, quality attributes affected (positively or negatively), and usage examples, while considering the justification for using a certain pattern and tradeoffs needs to be made, information that forms rationale described by a pattern's forces.

The *ASIP* presented in Table 11.1 helps improve the scenario development task, select suitable reasoning frameworks and increase confidence in the capabilities of architecture to satisfy particular quality sensitive

scenarios as a result of using certain patterns. For example, architects can identify suitable patterns by comparing the scenarios and quality attributes supported by different patterns with the ones required by the stakeholders. Moreover, the abstract scenarios extracted from patterns can be used to instigate stakeholders' thinking while developing quality sensitive scenarios or the abstract scenarios can be concretized to specify quality attributes for a particular system. We have found the template a promising way of capturing, using, and transferring ADK [3].

11.3.2 Modeling Architecture Knowledge and Rationale

We have developed a conceptual data model that identifies and defines the main architectural constructs and their relationships, which form ADK to support architecting process. A conceptual model is the first stage in the development of an automated system for storing DR which could help organizations to store and access ADK [30]. The DAta Model for Software Architecture Knowledge (DAMSAK) is a customizable model to characterize the data required to capture architecture knowledge and rationale.

Table 11.2 Sample data for various architectural constructs

generic quality attribute	flexibility/scalability (ASR entity)
abstract scenario	application shall instantly notify changes to the interested clients (Scenario entity).
abstract scenario	application shall be able to handle simultaneous notification requests from increased number of clients (Scenario entity).
architecture decision	event notification (Architecture Decision entity).
design option 1	publish scribe (Alternative entity).
design option 2	Java RMI (Alternative entity).
design pattern	publish on demand (Pattern entity).

We believe that the DAMSAK can help develop a repository of reusable ADK. To demonstrate the potential use of such a repository, we provide one example. One of the authors helped design an application which needed instantaneous event notification to unknown number of client tools (see [1] for details). The publish–scribe architecture pattern with a publish-on-demand design pattern was selected for this mechanism. However, the Java RMI was also considered, as it was decided it was not sufficiently scaleable for expected increase in the number of notification requests.

If there were a repository for ADK management, the architect or a knowledge engineer could have stored different architectural primitives as illustrated in Table 11.2. ADK can also be obtained from case studies such as described in [8, 17] or quality attribute sensitive design primitives

reported in [7]. Thus, a ADK repository will be an organizational memory analogous to engineers' handbooks, which consolidate best knowledge [4].

Having access to a repository of generic ADK enables designers to use the accumulated "wisdom" in different projects. For example, instantiating abstract scenarios into concrete ones, contextualizing design decisions and others. The project specific data model will also have other entities to capture and consolidate ADK and rationale that is specific to a project. For example, design history, findings of SA evaluation, architectural views of interest to each type of stakeholders and others. A project specific repository system will be populated with the specialized versions of data drawn from the organizational repository, standard work products of the design process, logs of the deliberations and histories of documentation [4].

Model Development Process and Model Description

The conceptual data model of architecture knowledge consists of primitives or semantic elements, which characterize the constructs and terminology used in designing and communicating architecture artifacts. To develop DAMSAK presented in Fig. 11.2, we used several approaches to arrive at an appropriate set of architectural constructs, namely:

- We read several textbooks on software architecture (e.g. [8, 11, 14, 17, 22]) and analyzed their examples and case studies to support our exploration of relevant architectural constructs and their attributes.
- Our work on comparison and classification of SA evaluation methods [2] was an important means of identifying relevant literature.
- We reviewed a selected set of gray literature (such as PhD theses and technical reports) in software architecture (e.g. [7, 10, 39]) and standards for documenting architectures [27].
- We reviewed the literature in other engineering disciplines and Human–Computer Interaction (e.g. [18, 24, 34, 44, 48] to discover appropriate constructs, which describe DR.

Our next step was guided by using the Unified Modeling Language for database design [37]. We assessed the model by reference to the literature, in particular [27, 39], which provide reference models for describing and evaluating SA. Following is a brief description of the data model.

The *Stakeholder* entity characterizes those people who have any kind of interest in the architecture process or product [39] such as developers, testers, managers, evaluators, maintainers, and many more [17]. This entity helps keep track of the people who contribute to or consult a knowledge base. Such information can be used to design a recognition program to motivate people to contribute or use an architecture knowledge repository.

Stakeholder
UserId: smallint
UserName: text
SkillSet*: text
FullName: text
EmailAddress: text

Scenario
ScenarioId: smallint
SDescription: text
ScenarioType*: smallint
Source: text
Justification: text
Status: char(10)
ScenarioHistory*: smallint
Keyword*: text

ASR
Id: smallint
ASRName: text
Description: text
Explanation: text
ProposedBy: text
Keyword*: text

Propose
Characterise
RelatedTo
DevelopedFor
Identify
SatisfiedBy
ExtractedFrom

AnalysisModel
ModelId: smallint
ModelName: text
Description: text
Parameters*: smallint
EquivalentModel: smallint
Rule*: smallint
Keyword*: text

EffectOfPattern
TypeOfEffect: char
Description: char
Rationale: char

SupportInformation
Explanation: char
Risks: char
NonRisks: char
Assumption: char
Constraint: char
Arguments*: int

Pattern
PatternId: smallint
PatternName: text
PatternType: text
Description: text
Context: text
Problem: text
Solution: text
Forces*: text
UsageExample*: text
Keyword*: text

EquivalentTo
Manipulates
FoundIn
ComposedOf
Make

DesignTactic
TacticId: smallint
TacticName: text
TacticRationale: text
TacticCategory: smallint
Rule: text
Keyword*: text

RelatedTa
Dependency
Applies
Uses
ChildOf
ConsistsOf

ArchitectureDecision
ADId: smallint
ADName: text
ADDescription: text
DecisionHistory*: smallint
Keyword*: text

DesignOption
AId: smallint
OName: text
ODescription: text

DesignRationale
Id:
BriefDescription: text
Comment: text
Constraint: text
Assumption: text
Strength: text
Weakness: text
Cost: text
Benefit: text
Complexity: text
UnresovledIssues*: smallint
Justification: text
Rule: text
Context: text
TradeOffs: text
Arguments*: smallint
OtherInformation: text

AttachedTo
For
RelatedAlternative
CapturedBy

ArchitectureDescription
Id:
Description: text
TypeOfDescription: text
ArchitectureView*: text
ViewType: text
RelatedView*: smallint

This link between DR and ArchitectureDecision reflects the fact that each decision also has DR attached to it. This DR cuts across the design options included in an architecture deicsion.

This link between DesignRationale and DesignOption reflects the fact that each design option shall have a rationale attached to it.

Fig. 11.2. Data model characterizing SA knowledge and rationale

Architecturally significant requirements (ASRs) are those requirements that have broad cross-functional implications. Such requirements are often Nonfunctional requirements (NFRs), also called Quality Attributes (QAs) [8, 39], but can also include functional aspects such as security functionality. This entity is used to describe and explain various aspects of an ASR. An ASR can be supported or hindered by one or more patterns used in a particular architecture decision. This is characterized by the *EffectOfPattern* association entity.

Scenario is a textual definition of an ASR. A scenario can be classified into different types of ASR such as availability, reliability, and modifiability [8]. A source attribute of a scenario describes whether the scenario has been elicited from a stakeholder or distilled from a pattern. A scenario has a history of changes made to it. An abstract scenario can help identify one or more analysis models to analyze design decisions.

Analysis Model is a reasoning framework that is used to systematically reason about the effect of different design tactics on required scenarios. A reasoning framework provides the vocabulary and analytical machinery for describing and deducing particular system properties. It consists of a set of independent and dependent parameters, their relationships and associated rules that need to be observed in evaluating the effects of a tactic [5].

Pattern characterizes known solution to a recurring problem in a particular context [22]. The term pattern denotes design pattern, architecture pattern, or architectural style. A pattern provides a mechanism for documenting and reusing design knowledge accumulated in terms of problem, solution, forces, and usage examples by experienced practitioners.

Tactic is a design mechanism for achieving the desired level of a QA by manipulating some aspect of an analysis model for that QA through design decisions [5]. A tactic may be classified into different categories of tactics, for example architectural, design, or implementation. A pattern may contain one or more tactics. A tactic is applied to satisfy one or more scenarios. This entity also captures the rationale for a tactic and any rules that should be observed to achieve the promised benefits of using that tactic.

Architecture Decision is a high-level design decision taken to satisfy a set of ASR. If we conceptualize the architecture design process as a decision making activity, an architecture decision is a choice among design options based on certain criteria [24]. A decision may have a history of the changes made to it along with any consequences of the changes on the other decisions. There may be interdependency between various decisions. For instance, an earlier decision may limit the options available or impose some constraints on the subsequent decisions. Any changes in a decision should consider the consequences for the dependency relationships.

Design option is a design decision that can be evaluated and selected to satisfy one or more functional or nonfunctional requirements [24]. Design decisions may be related to each other such that the selection of one design option requires the selection or rejection of another design option. A pattern may be used for one or more design options and a design option may apply one or more patterns. A design decision may use several tactics.

DR is the reason behind a design decision (architecture or option). DR consists of all the background information that may be used or generated during the decision making process. Such information is valuable to people who deal with the product of the decision making process [12, 24]. The DR associated with each design option records the required background knowledge essential to evaluate it with respect to other design options. The DR associated with an architecture decision cuts across all the design options selected for a particular architecture decision.

Effect of Pattern defines the effect (positive or negative) of a pattern on a particular ASR [14]. This entity captures the reason behind certain types of effect, information that forms the rationale. This is an association entity. There are a number of other association entities required to capture appropriate data: we show only the most important ones in Fig. 11.2.

Support information is an association entity that captures the background information to justify the choice of a specific architectural decision for a particular scenario. This background information includes explanation of the decision, risks considered, assumptions, and constraints. Such information is valuable for reusability of an architecture decision.

Architecture Description characterizes the data required to document an architecture according to certain standards or approaches (e.g. [16, 27]). An architecture description is usually organized into one or more views, which are models of architectural structures. Views can be categories into view types and architectural description should also capture the relationships among different views. The V&B [16] approach also emphasizes the need to capture information that cuts across several views. The view attribute of this entity is a complex attribute, which would be an entity in its own right in a fully normalized data model.

To the best of our knowledge, this data model represents a first systematic attempt to formally enumerate the architecture knowledge domain. There is always a tradeoff between the size and the representative ability of a data model as it is difficult to conceptualize a domain and tease it out to the level of entities, attributes, and relationships. Another source of difficulty in modeling architecture design knowledge is researchers and practitioners use different terms and describe architectures at various levels of abstractions. That is why we have developed a moderate size of data

model that can be tailored or extended to various organizations' needs, e.g., to support the "*architecture decision description templates*" described in [52], to implement an architecture knowledge repository like [38], or to develop an architecture design decision support system. Moreover, this data model can be easily modified to cater the data needs of software architecture reviews, which help capture architecture DR.

11.3.3 Empirical Assessment

The three techniques to capture implicit and explicit design knowledge described in Sect. 3 have been empirically evaluated in research labs or industrial settings [4, 25, 50]. However, their effectiveness needs to be empirically assessed in SA domain with real projects, something which we plan to achieve in the next step on the project. As mentioned earlier, the conceptual data model has been assessed by reference to published literature as is evident from the citation for each entity. Moreover, this data model incorporates the concepts described in the meta-models of IEEE standards for architecture description [27], architecture review context conceptual model of SARA report [39]. Furthermore, it can capture most of the data recommended for V&B approach of documenting architecture [16]. However, we do not claim entity-to-entity mapping.

To assess the effectiveness of the different components of the *pattern-mining* approach and the usefulness of the *ASIP* in architecture design and evaluation, we have designed and implemented an empirical research program consisting of an observational study and two controlled experiments. The observational study was aimed at finding out the average amount of time taken for mining patterns, the effectiveness of the *pattern-mining* process, guidelines, template to support the process, and the perception of the participants of the usefulness of the *ASIP*. The controlled experiments were designed to assess the value of *ASIP* during design and evaluation activities. The results of the observation study support the view that the *pattern-mining* process is effective and that the information obtained is useful [3], all the 18 subjects that replied to our first questionnaire found the proposed process and guidelines helpful in mining information from patterns. And 22 of the 24 subjects replying to subsequent questionnaires found the extracted information in templates more useful than standard pattern information for performing architecting activities. Objective quantitative data gathered during controlled experiments are yet to be analyzed, and we are expecting that the findings of the controlled experiments will be available in the near future.

11.4 Conclusions and Open Issues

One of the general conclusions is that recording the rationale for architecture decisions is an intuitively appealing idea, which has enormous potential benefits. However, there is little guidance and no support mechanism for capturing and managing ADK. In this chapter, we contribute to the growing efforts of software architecture rationale management by proposing a framework of three components to support the management of ADK. We hope that both researchers and practitioners can experiment with the ideas presented in this chapter and provide us some insights to refine our approach. In particular, we look forward to seeing how the proposed data model can be applied in practice to develop an organizational repository, which can be populated with ADK by following the "*pattern-mining*" process.

Despite continuous efforts by researchers and practitioners, including ours, the architecture community has a long way to go before ADK capture becomes a widely accepted practice. As Conklin [18] mentioned, successful transfer of any technology of capturing rationale will need to answer the questions like: what is the cost of not capturing and managing ADK, who is responsible for capturing and maintaining it, what are the incentives for the designers to take extra burden of documenting DR, and what are the mechanisms in place to prevent rationale being used for firing or prosecuting architects if a decision turns out to be an error? These are some of the issues that we plan to explore in our future efforts

Acknowledgments. The first author wishes to thank Len Bass, Antony Tang, and Mark Staples for useful comments on various aspects of the data model. National ICT Australia is funded through the Australian Government's Backing Australia's Ability initiative, in part through the Australian Research Council.

References

[1] Al-Naeem T, Gorton I, Ali-Babar M, Rabhi F, Benatallah B (2005) A quality-driven systematic approach for architecting distributed software applications. In: Proceedings of International Conference on Software Engineering, pp. 244–253

[2] Ali-Babar M, Zhu L, Jeffery R (2004) A framework for classifying and comparing software architecture evaluation methods. In: Proceedings of Australian Software Engineering Conference, pp. 309–318

[3] Ali-Babar M, Kitchenham B, Maheshwari P, Jeffery R (2005) Mining patterns for improving architecting activities – A research program and preliminary assessment. In: Proceedings of International Conference on Empirical Assessment in Software Engineering, 54–63
[4] Arango G, Schoen E, Pettengill R (1993) A process for consolidating and reusing design knowledge. In: Proceedings of International Conference on Software Engineering, pp. 231–242
[5] Bachmann F, Bass L, Klein M (2003) Deriving architectural tactics: A step toward methodical architectural design, Tech Report CMU/SEI-2003-TR-004, SEI, Carnegie Mellon University, USA
[6] Basili VR, Caldiera G (1995) Improving software quality reusing knowledge and experience. Sloan Management Review 37(1): pp. 55–64
[7] Bass L, Klein M, Bachmann F (2000) Quality attribute design primitives, Tech Report CMU/SEI-2000-TN-017, SEI, Carnegie Mellon University, USA
[8] Bass L, Clements P, Kazman R (2003) Software Architecture in Practice. 2 ed. Addison–Wesley
[9] Bass L., John B.E. (2003) Linking usability to software architecture patterns through general scenarios. Journal of Systems and Software 66(3): pp. 187–197
[10] Bengtsson P (2002) Architecture-level modifiability analysis Ph.D. Thesis Blekinge Institute of Technology, Sweden
[11] Bosch J (2000) Design and use of software architectures: Adopting and Evolving a Product-line Approach. Addison-Wesley
[12] Bosch J (2004) Software architecture: The next step. In: Proceedings of the European Workshop on Software Architecture, pp. 194–199
[13] Bratthall L, Johansson E, Regnell B (2000) Is a design rationale vital when predicting change impact? – A controlled experiment on software architecture evolution. In: Lecture Notes in Computer Science, F. Bomarius and M. Oivo (eds.) Springer, Berlin Heidelberg New York. pp. 126–139
[14] Buschmann F, Meunier R, Rohnert H, Sommerlad P, Stal M (1996) Pattern-oriented Software Architecture: A System of Patterns: Wiley, New York
[15] Carroll JM, Moran TP (1991) Introduction to the special issue on design rationale. Human–Computer Interaction 6: pp. 197–200
[16] Clements P, et al. (2002) Documenting software architectures: Views and Beyond: Addison-Wesley
[17] Clements P, Kazman R, Klein M (2002) Evaluating Software Architectures: Methods and Case Studies: Addison-Wesley
[18] Conklin J, Burgess-Yahkemovic KC (1991) A process-oriented approach to design rationale. Human–Computer Interaction 6(3–4): pp. 357–391
[19] Curtis B, Krasner H, Iscoe N (1988) A field study of the software design process for large systems. Communications of the ACM 31(11): pp. 1268–1287
[20] Dutoit AH, Paech B (2001) Rationale management in software engineering, in Handbook of Software Engineering and Knowledge Engineering, S. Change, (ed.): World Scientific Publishing, Singapore

[21] Dutoit AH, Paech B (2002) Rationale-based use case specification. Requirements Engineering 7(1): pp. 3–19
[22] Gamma E, Helm R, Johnson R, Vlissides J (1995) Design patterns-elements of reusable object-oriented software: Addison-Wesley, Reading, MA
[23] Gross D, Yu E (2000) From non-functional requirements to design through patterns. In: Proceedings of 6th International Workshop on Requirements Engineering Foundation for Software Quality
[24] Gruber T, Russell D (1991) Design knowledge and design rationale: A framework for representation, capture, and use, Tech Report KSL 90-45, Knowledge Laboratory, Stanford University, Stanford, USA
[25] Henninger S (2003) Tool support for experience-based software development methodologies. Advances in Computers 59: pp. 29–82
[26] Herbsleb JD, Kuwana E (1993) Preserving knowledge in design projects: What designers need to know. In: Proceedings of Human Factors in Computing Systems, 7–14
[27] IEEE (2000) Recommended practices for architecture description of software-intensive systems. IEEE Standard No. 1471
[28] John BE, Bass L, Sanchez-Segura MI, Adams RJ (2004) Bringing usability concerns to the design of software architecture. In: Proceedings of IFIP Working Conference on Engineering for Human–Computer Interaction, pp. 1–19
[29] Kazman R, Carriere SJ, Woods SG (2000) Toward a discipline of scenario-based architectural engineering. Annals of Software Engineering, 9(1–4): pp. 5–33
[30] Kitchenham BA, Hughes RT, Linkman SG (2001) Modeling software measurement data. IEEE Transactions on Software Engineering 27(9): pp. 788–804
[31] Kneuper R (2001) Supporting software processes using knowledge management, in Handbook of Software Engineering and Knowledge Engineering, S.K. Chang, (ed.). World Scientific Publishing, Singapore, pp. 579–606
[32] Kunz W, Rittel HWJ (1970) Issues as elements of information systems. W-P 131 Institute of Urban and Regional Development, University of California, Berkeley, USA
[33] Lee J (1991) Extending the Potts and Bruns model for recording design rationale. In: Proceedings of International Conference on Software Engineering, pp. 114–125
[34] Lee J, Lai KY (1991) What's in design rationale? Human–Computer Interaction 6(3–4): pp. 251–280
[35] MacLean A, Young RM, Bellotti VME, Moran TP (1991) Questions, options, and criteria: Elements of design space analysis. Human–Computer Interaction 6(3–4): pp. 201–250
[36] McCall R (1987) PHIBIS: Procedural hierarchical issue-based information systems. In: Proceedings of International Congress on Planning and Design Theory, pp. 17–22
[37] Naiburg EJ, Maksimchuk RA (2001) UML for database design: Addison-Wesley, Reading, MA

[38] Niemela E, Kalaoja J, Lago P (2005) Toward an architectural knowledge base for wireless service engineering. IEEE Transactions of Software Engineering 31(5): pp. 361–379
[39] Obbink H, et al. (2001) Software architecture review and assessment (SARA) report, Tech Report SARA W.G
[40] Parnas D, Clements P (1986) A rationale design process: How and why to fake it. IEEE Transactions of Software Engineering 12(2): pp. 251–257
[41] Pena-Mora F, Vadhavkar S (1997) Augmenting design patterns with design rationale. Artificial Intelligence for Engineering Design, Analysis and Manufacturing 11(2): pp. 93–108
[42] Perry DE, Wolf AL (1992) Foundations for the study of software architecture. ACM SIGSOFT, Software Engineering Notes 17(4): pp. 40–52
[43] Potts C, Burns G (1988) Recording the reasons for design decisions. In: Proceedings of International Conference on Software Engineering, pp. 418–427
[44] Potts C (1995) Supporting software design: Integrating design methods and design rationale. In: Design Rationale: Concepts, Techniques, and Use, J.M. Carroll (ed.). Lawrence Erlbaum Associates, Hillsdale, NJ. pp. 295–321
[45] Potts C (1999) Scenic: A strategy for inquiry-driven requirements determination. In: Proceedings of Symposium on Requirements Engineering, pp. 58–65
[46] Probst GJB (1998) Practical knowledge management: A model that works, in Prism. http://know.unige.ch/publications/Prismartikel.PDF, accessed on 1st July 2005
[47] Ramesh B, Dhar B (1992) Supporting systems development by capturing deliberations during requirements engineering. IEEE Transaction on Software Engineering 18(6): pp. 498–510
[48] Regli WC, Hu X, Atwood M, Sun W (2002) A survey of design rationale systems: Approaches, representation, capture and retrieval. Engineering with Computers 16: pp. 209–235
[49] Robillard PN (1999) The role of knowledge in software development. Communication of the ACM 42(1): pp. 87–92
[50] Skuce B (1995) Knowledge management in software design: A tool and a trial. Software Engineering Journal September: pp. 183–193
[51] Sutcliffe A (1995) Requirements rationales: Integrating approaches to requirement analysis. In: Proceedings of Symposium on Designing Interactive Systems, pp. 33–42
[52] Tyree J, Akerman A (2005) Architecture decisions: Demystifying architecture. IEEE Software 22(2): pp. 19–27